CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

Endbericht

12. Oktober 2006

Projektgruppe Syspect

Carl von Ossietzky Universität Oldenburg
Fakultät II
Department für Informatik
Abteilung Entwicklung korrekter Systeme

Inhaltsverzeichnis

Einleitung **1**

I Seminarausarbeitungen **3**

1 Eclipse **7**
1.1 Was ist Eclipse? . . . 7
1.1.1 Geschichte von Eclipse . . . 7
1.1.2 Grundlegende Eigenschaften von Eclipse . . . 7
1.1.3 Eclipse-Projekte . . . 8
1.2 Projektverwaltung in Eclipse . . . 8
1.2.1 Arbeitsbereich . . . 8
1.2.2 Projekte erzeugen . . . 8
1.2.3 Perspektiven in Eclipse . . . 9
1.2.4 Quellcode editieren . . . 11
1.2.5 Kompilieren und Debuggen . . . 11
1.2.6 Dokumentation . . . 12
1.3 Versionsverwaltung in Eclipse . . . 12
1.3.1 CVS-Befehle in Eclipse . . . 12
1.3.2 Erweiterte Funktionen . . . 13
1.3.3 Subversion in Eclipse . . . 13
1.4 Der Build-Prozess mit ANT . . . 13
1.4.1 Was ist ANT? . . . 13
1.4.2 Syntax von ANT . . . 13
1.4.3 ANT Beispiel . . . 14
1.4.4 ANT in Eclipse . . . 15

2 Eclipse-Plugins und GEF **17**
2.1 Einleitung . . . 17
2.1.1 Plug-ins und Eclipse . . . 17
2.1.2 Installation von Plug-ins . . . 18
2.2 Entwicklung eines Plug-ins . . . 19
2.2.1 Grundbausteine eines Plug-ins . . . 20
2.2.2 Ressourcen-Framework . . . 22
2.2.3 Editoren und Ansichten . . . 23
2.2.4 Perspektiven . . . 24

2.3 Features - Plug-ins verpacken . . . 24
2.4 Erweiterung durch Plug-ins - GEF . . . 25
2.5 Fazit . . . 26

3 Fujaba **27**
3.1 Einleitung . . . 27
3.2 Architektur von FUJABA . . . 28
3.3 Architektur von FUJABA4Eclipse . . . 30
3.4 Zusammenfassung mit Hinblick auf eine Weiterverwendung im Rahmen des Projektes . . . 31

4 ArgoUML **33**
4.1 Einleitung . . . 33
4.2 Allgemeines zu ArgoUML . . . 33
4.2.1 Benutzeroberfläche . . . 34
4.2.2 Funktionen von ArgoUML . . . 35
4.2.3 Kognitive Unterstützung: . . . 36
4.2.4 Dokumentation von ArgoUML . . . 37
4.3 Projektrelevante Themen . . . 37
4.3.1 Aufgaben der Projektgruppe Syspect . . . 37
4.3.2 ArgoUML Intern . . . 38
4.4 Handlungsbedarf der Projektgruppe: . . . 39
4.5 Fazit . . . 40

5 Community Z Tools **43**
5.1 Einleitung . . . 43
5.2 Werkzeuge . . . 44
5.3 ZML . . . 44
5.3.1 Aufbau . . . 45
5.4 JEdit-Plug-ins . . . 45
5.4.1 CZTSans . . . 45
5.4.2 ZCharMap . . . 45
5.5 Benutzung des Quellcodes . . . 46
5.5.1 Typecheck . . . 46
5.5.2 Lesen, Schreiben und Konvertieren von Z Darstellungen . . . 47
5.6 Lizenzen . . . 48
5.7 Fazit . . . 48

6 Einführung in UML und XMI **49**
6.1 UML - Unified Modeling Language . . . 49
6.1.1 Einleitung . . . 49
6.1.2 Übersicht über die einzelnen Diagrammtypen . . . 50
6.1.3 Drei Diagrammtypen am Beispiel erläutert . . . 51
6.1.4 Wesentliche Neuerungen in der UML 2.0 . . . 54
6.2 XMI - XML Metadata Interchange . . . 54
6.2.1 Einleitung . . . 54

6.2.2 Ein Überblick über XML . . . 54
6.2.3 Ein Beispiel . . . 55

7 CSP-OZ: Kombination zweier Methoden 57
7.1 Einführung . . . 57
7.2 Die Spezifikationssprache CSP . . . 57
7.2.1 Einleitung . . . 57
7.2.2 Syntax . . . 58
7.2.3 Informelle Semantik . . . 58
7.2.4 CSP-Semantiken . . . 62
7.3 Die Spezifikationssprache Z . . . 63
7.3.1 Z-Schema . . . 64
7.3.2 Transitionen . . . 64
7.4 Die Spezifikationssprache CSP-OZ . . . 65
7.4.1 CSP-OZ-Klassen . . . 66
7.4.2 Semantik . . . 66
7.5 Fazit . . . 67

8 CSP-OZ-DC und PEA 69
8.1 Einleitung . . . 69
8.2 Duration Calculus . . . 69
8.2.1 Syntax . . . 69
8.2.2 Semantik . . . 70
8.2.3 Beispiele . . . 71
8.3 Phase Event Automata . . . 72
8.4 CSP-OZ-DC . . . 72
8.4.1 CSP-Teil . . . 73
8.4.2 Object-Z-Teil . . . 73
8.4.3 DC-Teil . . . 73
8.4.4 Model-Checking . . . 74
8.4.5 Beispiel . . . 74
8.5 Zusammenfassung . . . 75

9 Ein UML-Profil für CSP-OZ-DC 77
9.1 Einleitung . . . 77
9.2 UML-Profil . . . 77
9.2.1 Capsule . . . 78
9.2.2 Protocol . . . 78
9.2.3 Übersetzung der Klassendiagramme . . . 78
9.2.4 Übersetzung der Statemachines . . . 79
9.2.5 Übersetzung der Strukturdiagramme . . . 79
9.3 Beispiel . . . 80
9.4 Zusammenfassung . . . 82

10 Software testen **83**
10.1 Einleitung . . . 83
10.2 Teststufen . . . 83
10.3 JUnit . . . 84
10.3.1 JUnit-Framework . . . 84
10.3.2 ANT: JUnit-Task . . . 88
10.4 Java Modeling Language . . . 89
10.4.1 Syntax . . . 89
10.4.2 Fehlerbehandlung . . . 91
10.4.3 jmlunit . . . 91
10.5 Zusammenfassung und Beurteilung . . . 92

11 Versionsverwaltung **93**
11.1 Einleitung . . . 93
11.1.1 Versionsverwaltung . . . 93
11.1.2 Versionierungsansatz . . . 94
11.2 Grundlegende Operationen . . . 94
11.2.1 Auschecken . . . 94
11.2.2 Einchecken . . . 95
11.2.3 Updaten . . . 95
11.2.4 Mergen . . . 96
11.3 Komplexere Operationen . . . 96
11.3.1 Umgang mit Dateien . . . 96
11.3.2 Umgang mit Verzeichnissen . . . 96
11.3.3 Ignore-Optionen und Properties . . . 97
11.4 GUIs . . . 97
11.5 Administration . . . 97
11.5.1 Repositories . . . 97
11.5.2 „Modul-Aliase“ . . . 98
11.5.3 Zugriffskontrolle . . . 98
11.5.4 Systemvorraussetzungen . . . 98
11.6 Fazit/Vergleich . . . 98

II Entwurfsdokumente **101**

12 Produktskizze **105**
12.1 Einleitung . . . 105
12.2 Die grafische Oberfläche der Entwicklungsumgebung . . . 105
12.2.1 Arbeiten mit Klassendiagrammen . . . 107
12.2.2 Arbeiten mit Strukturdiagrammen . . . 108
12.2.3 Arbeiten mit Zustandsdiagrammen . . . 109
12.3 Exportfunktionen der Entwicklungsumgebung . . . 109

13 Anforderungsdefinition **111**
13.1 Einleitung . . . 111
13.2 GUI-Darstellung der UML-Diagramme . . . 111
13.2.1 Metadaten . . . 112
13.2.2 Klassendiagramme . . . 112
13.2.3 Zustandsdiagramme . . . 115
13.2.4 Strukturdiagramme . . . 116
13.3 Nichtfunktionale Anforderungen . . . 118
13.3.1 Performance . . . 118
13.3.2 Sonstiges . . . 119
13.4 Dateiformate . . . 119
13.4.1 XML-Darstellung der CSP-OZ-DC-Spezifikationen . . . 119
13.4.2 XML-Darstellung von Phasen-Event-Automaten . . . 120
13.5 Anwendungsfälle . . . 122
13.5.1 Allgemeine Anwendungsfälle . . . 124
13.5.2 Anwendungsfälle in der Klassendiagrammansicht . . . 129
13.5.3 Anwendungsfälle in der Strukturdiagrammansicht . . . 133
13.5.4 Anwendungsfälle in der Zustandsdiagrammansicht . . . 138

14 Modifizierung der Anforderungen **143**
14.1 Einleitung . . . 143
14.2 Anpassung des UML-Profils . . . 143
14.3 Sonstige Änderungen . . . 145

15 Entwurf **147**
15.1 Einleitung . . . 147
15.2 Architektur . . . 147
15.2.1 Umgebung und Einbettung . . . 148
15.2.2 Plug-in-Struktur . . . 152
15.3 Feinentwurf . . . 157
15.3.1 Gliederung der Plug-ins . . . 157
15.3.2 Syspect Core Plug-in . . . 157
15.3.3 SyspectNavigator Plug-in . . . 163
15.3.4 SyspectPersistence Plug-in . . . 164
15.3.5 SyspectUtils Plug-in . . . 166
15.3.6 SyspectModel Plug-in . . . 167
15.3.7 SyspectModelConsistencyChecker Plug-in . . . 179
15.3.8 SyspectDiagramEditorBundle Plug-in . . . 180
15.3.9 Die drei Diagrammeditoren . . . 183
15.3.10 SyspectDiagramConsistencyChecker Plug-in . . . 188
15.3.11 Syspect OZ-DC-Util Plug-in . . . 189
15.3.12 SyspectProperties Plug-in . . . 190
15.3.13 SyspectCSPOZDCExport Plug-in . . . 194
15.3.14 SyspectPEAExport Plug-in . . . 201
15.3.15 SyspectImageExport Plug-in . . . 201
15.3.16 SyspectComponentTable Plug-in . . . 202

15.3.17 Richtlinien für den Quellcode 202

III Zusammenfassende Bewertung und Ausblick 205

16 Zusammenfassende Bewertung 207
16.1 Fallstudie *E-Mail-Client* 208

17 Ausblick 209
17.1 EMF 209
17.2 UML2 210
17.3 Druckfunktion 210
17.4 Notizzettelkommentare 211
17.5 TypeChecking 211
17.6 Verschieben von Klassen 212
17.7 Z-Text-Widget 212
17.8 AssemblyConnector ConnectionAnchor 212
17.9 Übersetzung nach PEA 213
17.10 Undo im OZ-Editor 213
17.11 Optimierung der Nutzung von javacc 213
17.12 Clipped Connections 214
17.13 *Drag and Drop* zwischen Projekten 214
17.14 Effizienteres Event-System 214
17.15 Verschieben von Regionen innerhalb von Zuständen 214
17.16 Definition der Methodentypen in der Konvertierung zu PEA 214
17.17 Performanz bei der Bendpoint-Berechnung 215
17.18 Test 215
17.18.1 Richtlinien für Tests 215

Anhang 219

Glossar 223

Index 226

Abbildungsverzeichnis 229

Literaturverzeichnis 232

Einleitung

Der vorliegende Endbericht beschreibt die Arbeit und Ergebnisse der Projektgruppe *Syspect* (**Sy**stem **Spec**ification **T**ool) der Abteilung *Entwicklung korrekter Systeme* an der Carl von Ossietzky Universität Oldenburg. Diese Einleitung gibt zunächst einen Überblick darüber, was eine Projektgruppe ist; anschließend wird auf die Ziele der Projektgruppe *Syspect* eingegangen. Es folgt ein Überblick über die Gliederung des gesamten Dokuments.

Projektgruppen

Die Studienordnung für den Diplom- bzw. Masterstudiengang Informatik an der Carl von Ossietzky Universität Oldenburg sieht für das vierte Studienjahr (bzw. erste Studienjahr für Masterstudenten) eine Projektgruppe vor. Diese besteht in der Regel aus sechs bis zwölf Teilnehmern. – *Syspect* besteht aus elf Studenten. – Sie sollen gemeinsam eine „substantielle Aufgabe“ im Umfang von 24 Kreditpunkten, also 4 Modulen bearbeiten. Neben der Entwicklung von Softwaresystemen, Hardwaresystemen oder einer Kombination aus beiden gehört zu einer Projektgruppe eine Seminarphase sowie die Anfertigung eines Endberichts.

Das Ziel einer Projektgruppe ist es, eine umfangreiche Aufgabe in Teamarbeit zu lösen. Dazu gehört es, Möglichkeiten der Organisation von Gruppenarbeit kennenzulernen und geeignete Kommunikationswege zu finden. Methoden des Software-Engineering können umgesetzt und somit Praxis-Erfahrungen gesammelt werden. Eine Projektgruppe kann unter Umständen auch Einblick in die aktuelle Forschungsarbeit der Universität geben. Kontakte zu Forschenden können geknüpft und Erfahrungen können ausgetauscht werden.

Die Projektgruppe beginnt mit einer Seminarphase, die kompakt an wenigen Tagen oder auf wöchentlichen Treffen stattfindet. – *Syspect* begann mit einer kompakten Seminarphase über zwei Tage. – In dieser Phase werden erarbeitete Grundlagen von den einzelnen Teilnehmern vorgestellt. Für die Arbeit im Verlauf des Jahres werden Sitzungstermine vereinbart, meist ein bis zwei wöchentlich. Auf diesen Sitzungen werden Ergebnisse der Arbeit zusammengetragen sowie die weitere Arbeit geplant. Die Vorgehensweise wird gemeinsam von den Studenten und Betreuern geplant. Eine Projektgruppe wird von einem oder mehreren Betreuern begleitet. Abgeschlossen wird die Projektgruppe mit dem Vorstellen der Ergebnisse auf dem Projektgruppentag des Fachbereichs Informatik.

Ziel der Projektgruppe

Ziel der einjährigen Arbeit der Projektgruppe war der Entwurf und die Implementierung einer grafischen Entwicklungsumgebung für eine formale UML-Teilsprache.
Es wurde ein UML-Werkzeug entwickelt, das die Modellierung entsprechend des UML-Profils für CSP-OZ-DC unterstützt. Die formale Methode CSP-OZ-DC kombiniert die

Prozessalgebra CSP [Hoa04] für die Spezifikation von Kommunikationsverhalten, die Methode Object-Z [Smi00] zur Beschreibung von Daten und Operationen und den Duration Calculus [ZHR91] zur Beschreibung von zeitlichen Anforderungen. Mit diesem Werkzeug können Komponenten-, Klassen- und Zustandsdiagramme gezeichnet werden. Für die Formalisierung der Daten ist die visuelle Eingabe der Spezifikationssprache Z möglich. Ebenso wird die Eingabe der Zeitaspekte durch Formeln des Duration Calculus (DC) unterstützt.
Das Werkzeug bietet eine Speicherung der Modelle in einem eigenen Datenformat. Neben einem Grafik- und einem LaTeX-Export kann die Spezifikation in Phasen-Event-Automaten übersetzt werden. Diese können von einem anderen Werkzeug gelesen werden, das wiederum ermöglicht, dass Model-Checker Eigenschaften der Spezifikation prüfen können.

Gliederung

Dieser Bericht ist wie folgt aufgebaut: Zunächst werden alle Seminarvorträge der Projektteilnehmer zu den in diesem Projekt relevanten Themen vorgestellt. Im Anschluss folgen die in der Gruppe erstellten Entwurfsdokumente, dabei handelt es sich im Einzelnen um folgende Kapitel:

Produktskizze: Die Produktskizze ist das Ergebnis der ersten Anforderungsanalyse und ist sowohl als Diskussionsgrundlage zwischen Kunden und Entwicklern gedacht als auch als Basis für die Anforderungsdefinition.

Anforderungsdefinition: Die Anforderungsdefinition ist eine Art Vertrag über das zu entwickelnde Produkt. Abweichungen hiervon müssen gesondert angegeben werden.

Modifizierung der Anforderungen: In diesem Kapitel werden die erwähnten Abweichungen von der Anforderungsdefinition beschrieben.

Entwurf: Getrennt in Architektur und Feinentwurf wird hier die technische Realisierung der *Syspect*-Applikation erläutert.

Schließlich wird in einem abschließenden Kapitel ein Fazit gezogen und ein Ausblick über in Zukunft zu ergänzende Features gegeben.

Teil I

Seminarausarbeitungen

In diesem Abschnitt werden alle Grundlagen, die für die Projektarbeit nötig waren, zum Nachschlagen zusammengetragen. Die Grundlagen wurden von einzelnen Personen erarbeitet und in Seminarvorträgen vorgestellt.

In der ersten Seminararbeit wird *Eclipse* vorgestellt, da dies eine weit verbreitete IDE zum Entwickeln von Applikationen ist, deren Funktionen den Projektgruppenteilnehmern vorgestellt werden sollen.
Da mit dem Werkzeug *Syspect* UML-Modelle grafisch entwickelt werden sollen, stellen die nächsten drei Ausarbeitungen Werkzeuge vor und untersuchen sie auf Erweiterungsmöglichkeiten und Wiederverwendbarkeit für *Syspect*.
Die UML-Modelle werden in die formale Spezifikationssprache CSP-OZ-DC übersetzt. Die Community-Z-Tools werden daher in einer Seminararbeit vorgestellt, da sie sich mit der Teilsprache Object-Z von CSP-OZ-DC befassen. Um in die UML 2.0 und das Speicherformat XMI einzuführen folgt eine Arbeit zu diesen Themen.
Die UML-Modelle werden durch ein UML-Profil für CSP-OZ-DC eingeschränkt. Drei Seminarausarbeitungen führen in die Spezifikationssprache, deren Darstellung als Phasen-Event-Automaten und das zughörige UML-Profil ein.
Anschließend folgt eine Arbeit über Testmethoden und eine Arbeit, die sich mit der Versionsverwaltung mit CVS und SVN auseinandersetzt.

Kapitel 1

Eclipse

Jürgen Happe

1.1 Was ist Eclipse?

Eclipse ist eine Open-Source-Gemeinde, deren Projekte sich auf das Bereitstellen einer erweiterbaren Entwicklungsplattform und eines Anwendungssystems für Softwareentwicklung konzentrieren. Eclipse bietet erweiterbare Werkzeuge und Systeme, die sich über den Lebenszyklus von Software, inklusive der Unterstützung für die Modellierung, Entwicklungsumgebungen für Java, C/C++ und andere Sprachen, Tests und Performanz, *Business Intelligence*, ergiebige Client-Anwendungen und eingebettete Entwicklung, erstrecken. Mit Eclipse ist allerdings in den meisten Zusammenhängen und so auch im folgenden Text das installierte Programm des Eclipse-Projekts gemeint und nicht die Open-Source-Gemeinde.

1.1.1 Geschichte von Eclipse

Das Eclipse-Projekt wurde 1999 von IBM- und OTI-Teams aus Nordamerika und Europa mit über 40 Vollzeit-Mitarbeitern gestartet. Es dauerte dennoch mehr als zwei Jahre, bis die Version 0.9 im Juni 2001 veröffentlicht wurde, der dann im Oktober desselben Jahres die Version 1.0 folgte. Im darauf folgenden Monat beschloss IBM, den Programmcode zu veröffentlichen, so dass Eclipse sich in ein Open-Source-Projekt verwandelte. Außerdem startete IBM eine Webseite [EFI05b], um Open-Source-Entwicklern eine Anlaufstelle zu bieten.

1.1.2 Grundlegende Eigenschaften von Eclipse

Eclipse wird als Plattform für Entwickler zunächst einmal dadurch attraktiv, dass es ein Open-Source-Projekt ist und auf Grund der Java-Architektur für jedes Betriebssystem zur Verfügung steht.

Die Eclipse-Entwickler haben es sich zum Ziel gesetzt, eine robuste, voll ausgestattete Industrieplattform für die Entwicklung hoch integrierter Werkzeuge zur Verfügung zu stellen. Verwirklichen wollen sie dies unter anderem durch eine Struktur, die nur die Plattform als Grundgerüst vorsieht und alle Funktionen als Plug-ins an die Plattform andocken lässt. Dadurch wird erreicht, dass man als Benutzer nie mehr als die gewünschten Funktionen installieren muss. Andere komplexere Entwicklungsumgebungen wirken aufgrund des Funktionsumfangs oft überladen und sind schwerer erweiterbar. Positiv bewerten lässt sich auch die zum Teil erreichte Unabhängigkeit von Programmiersprachen, da für viele beliebte Programmiersprachen ein Plug-in für Eclipse existiert oder entwickelt wird.

1.1.3 Eclipse-Projekte

Zu Eclipse gehören drei große Projekte, nämlich das *Eclipse Project*[EFIb], das *Eclipse Tools Project*[EFId] und das *Eclipse Technology Project*[EFIc].
Das *Eclipse Project* umfasst Plattform, *Java Development Tools* (JDT) und *Plug-in Development Environment* (PDE) und hat es sich zum Ziel gemacht, die Plattform und Grundgerüst zu überarbeiten und weiter zu entwickeln.
Das *Eclipse Tools Project* beinhaltet schlicht die wichtigsten Werkzeuge für Eclipse. Die Aufgabe ist die Koordination zwischen den Entwicklern, um Überschneidungen und Doppelungen bei Komponentenentwickelung zu vermeiden.
Das *Eclipse Technology Project* unterteilt sich in Forschungsprojekte, die sich mit Programmiersprachen und Entwicklungsumgebungen befassen, „Brutstätten“, die im kleinem Rahmen neue Funktionen für Eclipse entwickeln und Bildungsprojekte, die Lernhilfen und Fortbildungsmaterial zur Verfügung stellen.

1.2 Projektverwaltung in Eclipse

1.2.1 Arbeitsbereich

Der Arbeitsbereich von Eclipse kann unter der Menüoption *File->Switch Workspace* festgelegt werden. Grundsätzlich legt Eclipse alle Projekte, die im *Package Explorer* erstellt werden, auch in dem ausgewählten Verzeichnis ab, wobei die Verzeichnisstruktur beibehalten wird. Im Arbeitsbereich erstellt Eclipse außerdem versteckte Dateien, in denen sich Metadaten über Plug-ins, über die Projekte und den Arbeitsbereich insgesamt befinden. Mit der Menüoption *File->Import->Existing Projects into Workspace* ist es möglich, Projekte aus anderen lokalen Verzeichnissen in den aktuell gewählten Arbeitsbereich zu importieren.

1.2.2 Projekte erzeugen

Der *Project Wizard* (Abbildung 1.1) kann in Eclipse abhängig vom Installationsumfang Projekte erzeugen. Wenn zum Beispiel die *C Development Tools* (CDT) installiert sind, lassen sich neben den Java-Projekten auch C/C++-Projekte erzeugen. Neben dem Erzeugen von neuen Projekten gibt es auch noch Funktionen, die Projekte mit

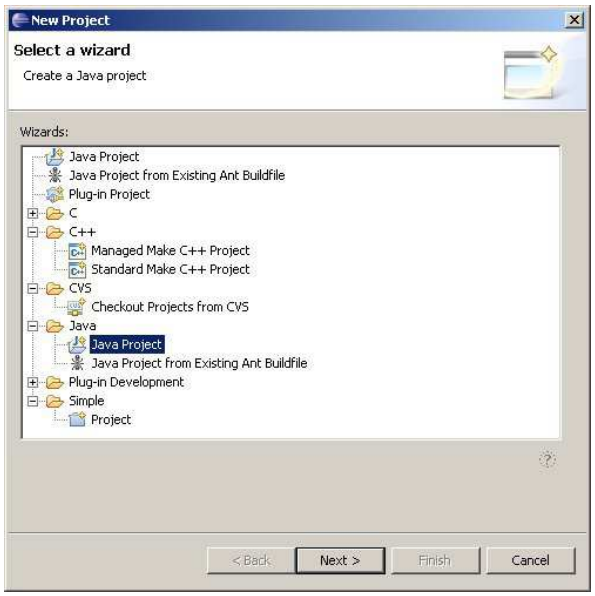


Abbildung 1.1: Project Wizard

Hilfe von ANT-Buildfiles generieren und das Auschecken von Projekten aus dem CVS ermöglichen. Außerdem kann der *Project Wizard* festlegen, welche Compilerversion benutzt werden soll und welche Bibliotheken in das Projekt eingebunden werden.

1.2.3 Perspektiven in Eclipse

Perspektiven in Eclipse ermöglichen es dem Benutzer, sich Projekte aus unterschiedlichen Ansichten anzusehen und sogar, Ansichten für seine individuellen Bedürfnisse zu erstellen. Konkret gibt es zum Beispiel eine Java-Perspektive (Abb.2), in der standardmäßig links der *Package Explorer*, in der Mitte die zu editierende Datei, unten eine Ausgabekonsole und rechts eine Liste aller Methoden und Variablen der Klasse erscheint. Diese Perspektive soll es dem Nutzer ermöglichen, viele relevante Informationen für das Bearbeiten und Kompilieren in der Übersicht zu behalten. Die *Java Browsing*-Perspektive zeigt im oberen Bereich vier Fenster, die nebeneinander Java-Projekte, -Pakete, -Klassen und -Methoden in strukturierter Weise darstellen. Das Ziel dieser Perspektive ist es, im Unterschied zur normalen Java-Perspektive, eine Übersicht über die gesamte Struktur des Projektes zu schaffen. Perspektiven zeigen also immer an den Kontext angepasste Blickwinkel auf ein Projekt.
Zu Perspektiven gehören aber nicht nur unterschiedliche Ansichten auf Java-Projekte. Es gibt auch eine eigene Ansicht für C/C++-Projekte, in der auf die Bedürfnisse von C-Entwicklern eingegangen wird, in dem ein zusätzliches *Make Targets*-Fenster erscheint und die Shortcuts verändert werden. So werden die Shortcuts, die in der Java-Perspektive zum Erstellen von Klassen und Paketen benutzt werden, ersetzt durch Shortcuts, die C/C++-Klassen und -Quelldateien erstellen.

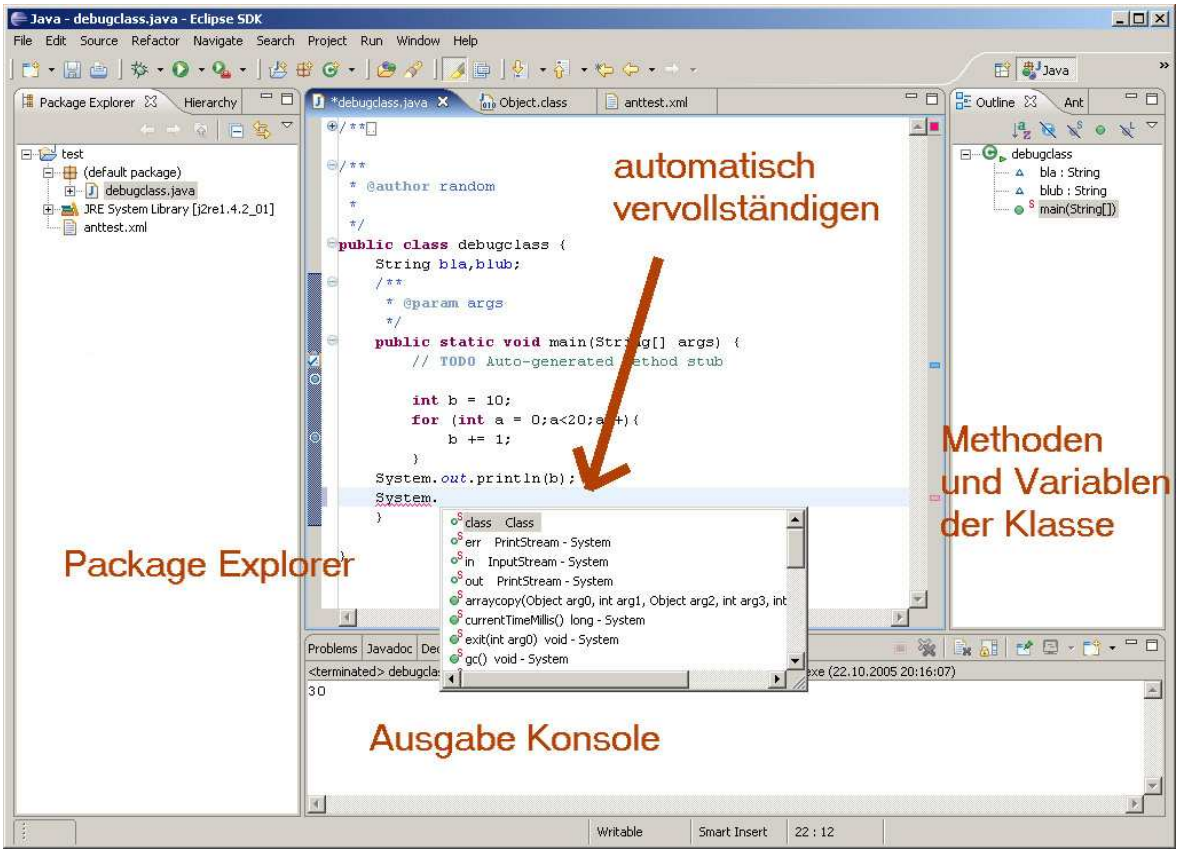


Abbildung 1.2: Java-Perspektive

1.2.4 Quellcode editieren

Unabhängig von der Perspektive lässt sich jede Datei in Eclipse aus dem Dateikontextmenü mit verschiedenen Editoren öffnen. Eine Ansicht auf die Dateien bieten dabei nicht nur die Editoren für die verschiedenen Programmiersprachen mit unterschiedlichem Syntax-Highlighting, sondern auch zum Beispiel der Webbrowser für Html-Dateien.
Eine weitere nützliche Funktion von Eclipse ist das automatische Vervollständigen von Ausdrücken. So wird der Java-Ausdruck `System.out.p` zu `System.out.println()`. Wenn ein Benutzer im ANT-Editor `tar` tippt, lässt der Ausdruck sich zu `<target name = ' '></target>` vervollständigen. Das Vervollständigen von Ausdrücken erfolgt über einen durch „Strg + Space“ aufrufbaren Dialog, der alle möglichen logischen Erweiterungen zum bereits getippten Ausdruck angibt.
Eclipse überprüft außerdem noch während der laufenden Eingabe den Programmcode auf Korrektheit. Der Nutzer kann also, ohne die Datei abzuspeichern und auszuführen, durch die von Eclipse markierten Zeilen Fehler finden.

1.2.5 Kompilieren und Debuggen

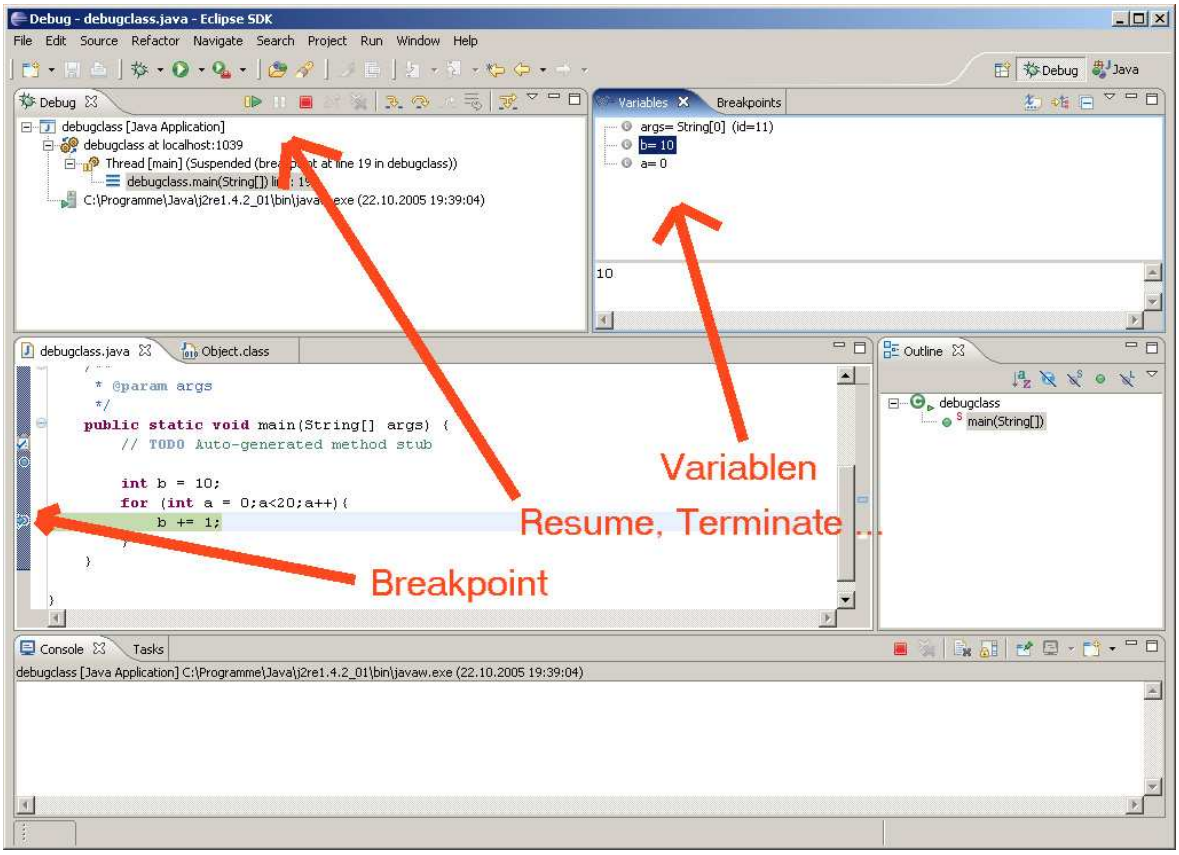


Abbildung 1.3: Debug-Perspektive

Unter den Shortcuts *Run* und *Debug* verbirgt sich die Möglichkeit, den Compiler mit verschiedenen Parametern und Variablen zu füttern und unterschiedliche Compiler-

konfigurationen für dasselbe Projekt zu erstellen. Am Besten debuggen kann man in Eclipse in der dafür vorgesehenen Perspektive (Abbildung 1.3). Breakpoints lassen sich allerdings überall durch einen Rechtsklick auf den linken Rand eines Editors setzen. Setzt man einen einfachen Zeilen-Breakpoint, hält der Debugger an, wenn das Programm in der Zeile ankommen ist. An dieser Stelle kann der Benutzer mit einem Klick in die Shortcutleiste des 'Debug'-Fensters das Programm einfach fortsetzen, es abbrechen, oder im 'Variablen'-Fenster Werte verändern, um so einen bestimmten Ablauf des Programms zu simulieren. Neben Breakpoints gibt es in Eclipse noch so genannte Watchpoints, die eine Anwendung immer dann stoppen, wenn der Wert eines Ausdrucks sich verändert, unabhängig davon, wo dies passiert. Watchpoints werden über den Menüpunkt *Run->Toggle Watchpoint* gesetzt, wobei man zuerst die gewünschte Variable markieren muss.

1.2.6 Dokumentation

Das Erzeugen von Dokumentation für Java-Projekte unterstützt Eclipse mit dem Menübefehl *Project->Generate Javadoc*. Hier lassen sich eine Reihe Einstellungen über das Aussehen der Dokumentation, wie zum Beispiel die Ausgabe von Navigationsleisten oder die Beachtung einzelner Javadoc-*Tags* (z.B. `@author`) festlegen. Alle diese Einstellungen lassen sich auch in einem ANT-Skript speichern, das nur ein einziges *Target* enthält, in welchem der Befehl `javadoc` mit allen im grafischen Dialog festgelegten Optionen eingetragen ist.

1.3 Versionsverwaltung in Eclipse

CVS ist im Standardpaket von Eclipse enthalten. Für Subversion kann das Plug-in Subclipse[tig] einfach und kostenlos herunter geladen werden.

1.3.1 CVS-Befehle in Eclipse

Zum Auschecken von Projekten stellt Eclipse die Funktion *File->Import->Checkout Project from CVS* zur Verfügung. Genau wie beim Kommandozeilenbefehl muss Host, Pfadname, Protokoll, Login, Passwort und der Name des Moduls angegeben werden. Danach wird das Modul als Projekt im Eclipse-Arbeitsbereich untergebracht.
Ist ein Projekt mit einem Repository verbunden, stehen im Dateikontextmenü, unter dem Punkt *Team*, die Funktionen *commit* und *update* zur Verfügung. Markiert man im *Package Explorer* jetzt ein oder mehrere Dateien oder Verzeichnisse, besteht die Möglichkeit, die neueste Version der Dateien oder Verzeichnisse aus dem Repository mit dem *update*-Befehl herunter zu laden. *Commit* lädt unter Angabe eines Kommentars über die Änderungen die markierten Dateien ins Repository hoch.
Soll eine neue Datei dem Repository hinzugefügt werden, kann man diese einfach im lokalen Arbeitsbereich mit dem *Package Explorer* erstellen und mit *commit* hochladen. Zum Löschen von Dateien sollte auf den *Repository Explorer* zurückgegriffen werden, da das Löschen einer Datei mit dem *Package Explorer* diese nur aus dem lokalen Arbeitsbereich entfernt.

1.3.2 Erweiterte Funktionen

Neben den Standardfunktionen wie *checkout* und *commit*, die im vorherigen Abschnitt erläutert wurden, hat Eclipse auch noch weitere erwähnenswerte Funktionen.
Der *Compare Viewer* ermöglicht es dem Benutzer, seine aktuell bearbeitete Datei mit der im Repository zu vergleichen. Das ist vor allem dann nützlich, wenn die Datei im Repository ebenfalls verändert wurde und die Änderungen zusammengeführt werden müssen. Beide Dateien werden nebeneinander angezeigt und Unterschiede werden markiert.
Der *Repository Explorer* kann dem Benutzer das gesamte Repository anzeigen und macht es möglich, jede einzelne Datei jeder Version mit ihrer Versionsgeschichte unter die Lupe nehmen.
In der *Team Synchronizing*-Perspektive hat der Entwickler einen Überblick über die einzelnen Dateien des gesamten Projekts, mit allen gerade eingehenden Änderungen, ausgehenden Änderungen und Konflikten.

1.3.3 Subversion in Eclipse

Zwischen Subversion und CVS gibt es in Eclipse im Frontend so gut wie keine Unterschiede. Alle Dialoge und Fenster sind sehr ähnlich, was den Vorteil hat, dass jeder, der CVS kennt, auch Subversion in Eclipse benutzen kann. Subclipse kann wie alle Plug-ins unter dem Menüpunkt *Help->Software Updates->Find and Install* installiert werden. Anschließend wählt man selbstverständlich den Menüpunkt *Search for new features to install* und trägt dann im nächsten Dialogfenster unter *Add new Remote Site* die Quelle 'http://subclipse.tigris.org/update'.

1.4 Der Build-Prozess mit ANT

1.4.1 Was ist ANT?

Der Name ANT[Apa06] entstand durch die Kurzbeschreibung des Tools „**A**nother **N**eat **T**ool“, was übersetzt bedeutet: „ein weiteres hübsches Werkzeug“. Konkreter verbirgt sich hinter ANT ein Java-basiertes Build-Tool. Das Ziel der Entwickler war eine Art Make [FSF06a], ohne die Make Schönheitsfehler. Die Build-Files von ANT sind XML-basiert und damit strukturiert, aber gelegentlich auch umständlicher als Make-Build-Files. ANT-Build-Files können in Eclipse editiert, mit einem dafür vorgesehenen Editor bearbeitet und als *External Tool* ausgeführt werden.

1.4.2 Syntax von ANT

Drei wichtige syntaktische Bausteine von ANT kurz erläutert:
<project name=“example“ default=“doc“ basedir=“.“></project>
Im Attribut *name* wird der Name festgelegt, auf den sich das Projekt bezieht. Der Wert von *default* gibt das standardmäßig auszuführende *Target* an. Das *basedir* ist der Verzeichnispfad, wo das Projekt im Verhältnis zur Build-Datei zu finden ist.

<property name="build" location="bin/bla/blub"/>
Durch die Angabe einer *property* wird es möglich, längere Verzeichnisangaben, die öfter in der Datei vorkommen, durch kürzer Ausdrücke zu ersetzen. In diesem Fall ließe sich der Pfadname `bin/bla/blub` durch `${build}` ersetzen.

<target name="compile" depends="init" description="compile the source">
<mkdir dir="bin"/> </target>
Targets führen die eigentlichen Befehle, wie zum Beispiel *<mkdir dir="bin"/>*, aus. Durch *depends* kann das *Target* von einem anderen *Target* abhängig gemacht werden, was bedeutet, dass das *Target*, von dem es abhängig ist, zuerst ausgeführt wird. Außerdem wird ein *Target* nicht ausgeführt, wenn das, von dem es abhängig ist, fehlschlägt.

1.4.3 ANT Beispiel

<project name="example" default="doc" basedir=".">

<target name="init">
<mkdir dir="bin"/>
</target>

<target name="build" depends=" init">
<javac srcdir="src" destdir="bin"/>
</target>

<target name="doc" depends="build ">
<javadoc sourcepath="." packagenames="src" destdir="doc">
</target>

</project>

Dieses Beispiel zeigt drei *Targets*, von denen das erste ein Verzeichnis erstellt, das zweite den Quellcode kompiliert und das dritte die Dokumentation erzeugt. So, wie das Programm jetzt aufgebaut ist, würden alle drei *Targets* in der eben erwähnten Reihenfolge abgearbeitet werden. Grund dafür ist, das ein *Target* erst das *Target* ausführt von dem es ggf. abhängig ist. Es wäre zwar auch möglich, alle drei Befehle in ein *Target* zu schreiben, dennoch ist die Trennung solcher Befehle in verschiedene *Targets* oft geschickter.
In dem Fall, dass das Verzeichnis nicht erzeugt werden kann, würde auch der Kompiliervorgang gar nicht erst starten. Ebenso würde eine Fehlermeldung des Compilers dazu führen, dass das letzte *Target*, also das Erzeugen der Dokumentation, nicht mehr gestartet wird. Ob das in diesem Fall sinnvoll ist, ist nicht unbedingt sicher. Es liegt aber auf der Hand, dass sich auf diese Weise klare logische Abhängigkeiten definieren lassen, wodurch sich gerade bei größeren Projekten Fehler vermeiden lassen.

1.4.4 ANT in Eclipse

Alle XML-Dateien können in Eclipse auch mit dem ANT-Editor geöffnet werden. Über die *External Tools* lassen sich ANT-Build-Dateien ausführen und es lässt sich vor allem genau festlegen, welche *Targets* ausgeführt werden sollen.

Kapitel 2

Eclipse-Plugins und GEF

Christian Günther

2.1 Einleitung

Im Rahmen der Projektgruppe Syspect entstand an der Universität Oldenburg diese Ausarbeitung, die die Entwicklung von Plug-ins für Eclipse 3.1 sowie die Benutzung des Plug-ins GEF beinhaltet. Ziel der Projektgruppe ist der Entwurf und die Implementierung einer grafischen Entwicklungsumgebung für eine formale UML-Teilsprache. Eclipse und GEF bieten sich dabei als Plattform für eine solche Entwicklungsumgebung an.

2.1.1 Plug-ins und Eclipse

Ein Plug-in (engl. *to plug in* – einstöpseln, anschließen) ist eine Softwarekomponente, welche die Funktionalität einer anderen Software benutzt und/oder erweitert. Es kann daher auch als Ergänzungs- oder Zusatzmodul beschrieben werden. Der Vorgang der Erweiterung muss dabei über festgelegte Schnittstellen der Software definiert werden, die erweitert werden soll. Diese Schnittstellen bilden in Eclipse den zentralen Mechanismus, um Funktionalitäten hinzuzufügen. Dementsprechend bildet ein Plug-in auch die kleinste Funktionseinheit in Eclipse.
Eclipse selbst ist „nur“ eine Allzweck-IDE und eine Integrationsplattform für Software-Werkzeuge. Die einzige Funktionalität beruht auf einem Laufzeitmodul der Eclipse-Plattform, das Plug-ins verwaltet und daher selbst kein Plug-in ist [Sha04].
Die Plattform bietet für Plug-ins unter anderem die Vorteile einer einheitlichen Umgebung sowie einer einfachen Integrationsmöglichkeit. Weiter gibt es keine Inkompatibilitäten. Damit die Vorteile gewährleistet werden können, stellt Eclipse verschiedene Bibliotheken, Schnittstellen und Frameworks zur Verfügung, auf die später genauer eingegangen werden soll.
Abbildung 2.1 verdeutlicht den Aufbau von Eclipse. Die als Eclipse (*The Eclipse Project*) bekannte IDE setzt sich aus der Eclipse-Plattform, deren Komponenten sowie

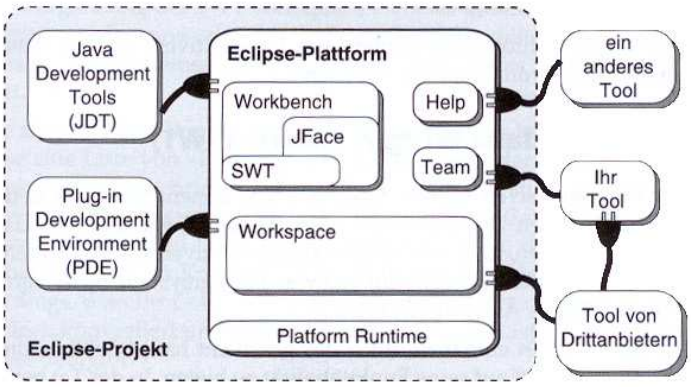


Abbildung 2.1: Struktur von Eclipse [Sha04]

dem JDT und PDE zusammen. Das Plug-in JDT zeigt die Entwicklungsumgebung für Java auf. PDE stellt ein Plug-in für die Plug-in-Entwicklung dar. Aufbauend auf der Plattform können weitere Plug-ins erstellt werden sowie Plug-ins auf Plug-ins aufbauen bzw. untereinander abhängig sein.
Der Plug-in-Mechanismus von Eclipse hat bei vielen Entwicklern und Anwendern Zuspruch gefunden. Entwickelt wurde Eclipse im Hause IBM, jedoch gibt es eine Reihe an bekannten Firmen, die mit bzw. für Eclipse entwickeln. Beispielhaft können an dieser Stelle HP, Borland, SAP und Oracle genannt werden [EFI05b]. Aufgrund der Tatsache, dass Plug-ins auf Plug-ins aufbauen können, erstreckt sich die Funktionalität von Plug-ins über etliche Bereiche. Plug-ins lassen sich beispielsweise zu den Kategorien Editoren, Grafik, Datenbanken, *Rich Client Platform*, Netzwerke, und Unterhaltung über zahlreiche Portale finden (z.B. [EFI06a], [EFI06b], [EFI05b]). Obwohl Eclipse selbst Open-Source ist, gibt es neben Open-Source-Plug-ins auch kommerzielle Plug-ins. Zudem es ist auch möglich, Plug-ins als *Rich Client Application* laufen zu lassen.

2.1.2 Installation von Plug-ins

Plug-ins lassen sich komfortabel über einen integrierten *Update Manager* in Eclipse installieren und aktualisieren. Jedoch kann das Plug-in auch in das Installationsverzeichnis entpackt werden. Wie Abbildung 2.2 zu entnehmen ist, befindet sich der *Update Manager* in der Menüleiste von Eclipse unter dem Punkt *Help->Software Updates*. Durch die Angabe einer *Update Site* ist es dem *Update Manager* möglich, gefundene Plug-ins aufzulisten und zu installieren. Die *Update Site* kann sowohl ein lokales Verzeichnis als auch eine URL im Netz sein. Eine weitere nützliche Funktion des *Update Managers* ist die Prüfung auf Abhängigkeiten verschiedener Plug-ins untereinander, d.h., entsprechende Inkompatibilitäten zwischen den Plug-ins werden direkt aufgezeigt. Das Installationsverzeichnis von Eclipse gliedert sich in die vier Unterverzeichnisse `configuration`, `features`, `plugins` und `readme`. Das Verzeichnis `plugins` enthält alle Plug-ins, die zur Eclipse-Installation installiert worden sind sowie „interne“ Plug-ins, welche zu Eclipse gehören. Beispielhaft kann hier die Plattform als internes Plug-in genannt werden. Ein Plug-in kann auch durch Kopieren in das Verzeichnis installiert werden. Das gilt ebenso für *Features* (siehe Abschnitt 2.3), welche sich dementsprechend in dem

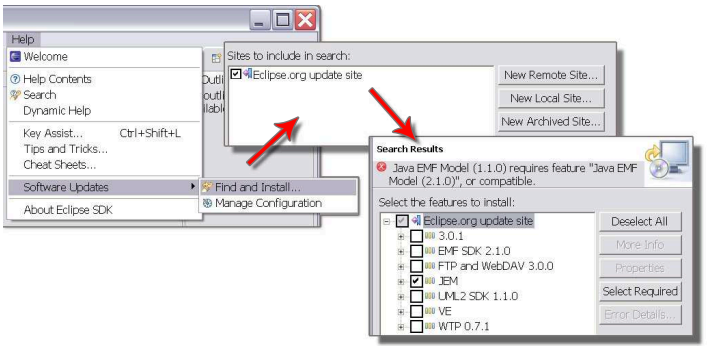


Abbildung 2.2: Screenshots des Update-Managers

`feature`-Verzeichnis befinden. Das Verzeichnis `configuration` enthält eclipse-spezifische Einstellungen, während das Verzeichnis `readme` weiterführende Informationen zu Eclipse bereitstellt.

2.2 Entwicklung eines Plug-ins

Da Eclipse in Java geschrieben ist, wird auch für Plug-ins diese Programmiersprache verwendet. Eclipse stellt durch das *Java Development Plugin* (JDT) einen sehr guten Java-Editor zur Verfügung. Aus diesem Grund bietet sich die Entwicklung von Plug-ins mit Eclipse an. Das JDT bietet allerdings keine Unterstützung für den Plug-in-Mechanismus. Deswegen gehört standardmäßig das Plug-in *Plug-in Development Enviroment* (PDE) auch zum *Eclipse Project*, welches eine direkte Plug-in-Entwicklungsumgebung für Eclipse-Plug-ins bietet. Es können somit auf komfortable Weise Eclipse-Plug-ins mit Eclipse selbst erstellt werden.

Die Entwicklung eines Plug-ins für Eclipse unterscheidet sich von dem Entwicklungsprozess von anderer Software. Der Hauptpunkt dabei ist, dass nicht von Grund auf nach einem Ansatz gesucht wird, die Software zu realisieren, sondern nach einem vorhandenen Ansatz in Eclipse, der erweitert werden kann [Sha04]. Mit Ansatz ist hier etwa ein Integrationspunkt für ein Plug-in bzw. eine Erweiterung gemeint. Das Plug-in wird somit auf die Plattform von Eclipse aufgebaut, integriert. Damit solche Integrationspunkte gefunden werden können, müssen die zur Verfügung gestellten Funktionen bzw. Schnittstellen, Bibliotheken und Frameworks bekannt sein, die solche Integrationspunkte zur verfügung stellen. Diese sollen an dieser Stelle kurz erläutert werden (vergleiche Abbildung 2.1 auf Seite 18).

SWT steht für *Standard Widget Toolkit* und ist eine Bibliothek für grafische Benutzeroberflächen. SWT greift auf die nativen Steuerelemente des Betriebssystems zurück und bildet somit die unterste Schicht im Gesamt-Framework der Benutzeroberfläche. SWT wurde auf eine Vielzahl von Fenster- und Betriebssystemen portiert und läuft auf allen nahezu äquivalent. Die Bibliothek ist nicht an Eclipse gebunden und kann daher auch außerhalb von Eclipse benutzt werden [Sha04].

JFace ist ein Benutzeroberflächen-Framework für allgemeine Benutzeroberflächenaufgaben. JFace setzt auf SWT auf bzw. erweitert dieses. Es enthält sowohl Frameworks für Widgets, Bild- und Schriftregistrierungen, Dialogfelder und Assistenten als auch für Aktionen und Erweiterungen [Sha04].

Die Workbench repräsentiert die Benutzeroberfläche der Eclipse-IDE. Sie gibt somit das Layout der Oberfläche vor und strukturiert bzw. verwaltet Editoren und Ansichten. Die Workbench benutzt dabei SWT und JFace [Sha04].

Die Workspace API and Resources unterstützen den Zugriff auf den von Eclipse verwendeten Workspacebereich bzw. bildet diesen intern in Eclipse ab.

Team Support bietet die Möglichkeit einer Versionierung und Konfiguration von Eclipse-Projekten und -Daten bzw. unterstützt die Integration von Repository-Systemen.

Das Hilfesystem für Eclipse schafft eine Basis für die Dokumentation eines Plug-ins. Das Framework unterstützt dabei sowohl die integrierte Hilfedokumentation, die kontextabhängige Hilfe, die Übersetzung von Hilfseinhalten in mehrere Sprachen, die Suchfunktion als auch ein eigenständiges Hilfesystem, welches außerhalb von Eclipse benutzt werden kann. Zudem gibt es eine aktive Hilfe, mit der bestimmte Aktionen für einen Benutzer aufgeführt werden können. Beispielhaft kann an dieser Stelle der Perspektivenwechsel genannt werden [Sha04].

2.2.1 Grundbausteine eines Plug-ins

Im Abschnitt 2.2 wurden so genannte Integrationspunkte angesprochen, auf die Plug-ins aufbauen können. Diese Integrationspunkte bilden den Grundbaustein des Eclipse-Plug-in-Mechanismus. Bei der Entwicklung eines Plug-ins muss daher bekannt sein, wo das Plug-in in Eclipse von technischer Seite integriert bzw. auf welche Grundfunktionen es aufgebaut werden kann. Dieses Prinzip wird durch so genannte *extension points* und *extensions* verwirklicht. *Extension points* definieren die Integrationspunkte, auf denen neue Plug-ins durch eine *extension* auf diesen *extension point* aufbauen können. Die *extension points* beziehen sich dabei auf die in Abschnitt 2.2 zuvor beschriebenen unterschiedlichen Schnittstellen bzw. die Schnittstellen bieten verschiedene *extension points* für Plug-ins an. Die *extensions* eines Plug-ins werden in einer XML-Datei mit dem Dateinamen `Plugin.xml` definiert (siehe Abbildung 2.4). Allein die Definition einer *extension* in der XML-Datei reicht für Eclipse aus, diese zu „laden“. So wird beispielsweise ein Menüeintrag ohne jegliche Implementierung bereits angezeigt. Die eigentliche Implementierung zu einer *extension*, welche das Plug-in bzw. die Erweiterung im Plug-in eigentlich ausmacht, wird durch eine Java-Klasse repräsentiert, die eine von dem *extension point* vorgegebene Schnittstelle implementiert. Diese Klasse wird erst beim direkten Gebrauch von Eclipse geladen, um Ressourcen einzusparen. Somit kann gewährleistet werden, dass auch bei einer großen Menge an installierten Plug-ins Eclipse noch lauffähig bleibt. Die Referenzierung auf die Klasse, welche die Schnittstelle des

extension point implementiert, befindet sich ebenfalls in der *extension*-Definition in der `Plugin.xml`. Da Plug-ins auch auf andere Plug-ins aufbauen können, definieren Plug-ins auch gegebenenfalls selbst *extension points*.

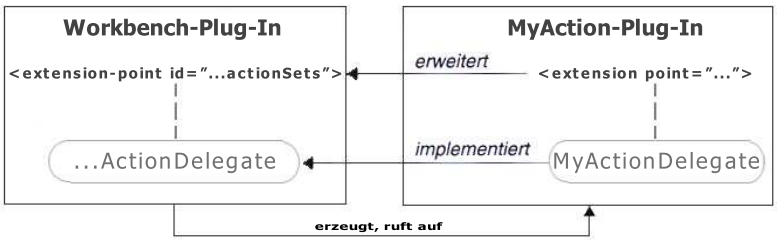


Abbildung 2.3: extension-point und extension

In der Abbildung 2.3 stellt das *Workbench*-Plug-in den `extension-point actionSets` zur Verfügung, durch diesen sich im *Workbench*-Menü eine neue *Action* hinzufügen lässt. Diese neue *Action* wird durch `MyAction-Plugin` bzw. durch die *extension* auf diesen *extension point* definiert. Damit das *Workbench*-Plug-in auf Anfrage die neu hinzugefügte Aktion ausführen bzw. aufrufen kann, muss `MyActionPlugin` die von dem *Workbench*-Plug-in vorgegebene Schnittstelle `ActionDelegate` implementieren. Die folgende Abbildung 2.4 zeigt dieses Szenario in Verbindung mit der `Plugin.xml` und der zu implementierenden Klasse.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
    <extension point="org.eclipse.ui.actionSets">
        <actionSet
            id="HelloWorld.actionSet"
            label="My Action Set"
            visible="true">
            <menu
                id="MyMenu"
                label="My Menu">
            </menu>
            <action
                class="helloWorld.actions.MyAction"
                id="helloWorld.actions.MyAction"
                label="My Action"
                menubarPath="MyMenu/"/>
        </actionSet>
    </extension>
</plugin>
```

```
public class MyAction implements IWorkbenchWindowActionDelegate {
    ...
    public void run(IAction action) {
```

Abbildung 2.4: Plugin.xml und Implementierung

Bevor die *extensions* des Plug-ins definiert und implementiert werden können, muss das Plug-in selbst definiert werden. Dies geschieht durch ein Plug-in-Manifest in Form einer Datei mit dem Namen `MANIFEST.MF` (siehe Abbildung 2.5). Eine der wichtigsten Eigenschaften des Manifests ist die Definition der Abhängigkeiten zu anderen Plug-ins. Eclipse kann beim Start anhand der Manifeste eine interne Plug-in-Registry erstellen und die Plug-ins in Abhängigkeiten laden [Sha04]. Aktiviert wird ein Plug-in allerdings nur dann, wenn es gebraucht wird – es sei denn, das Plug-in erzwingt ein frühes Aktivieren. Zu diesem Manifest gehört, wie auch bei den *extensions*, eine Java-Klasse, die durch das Manifest referenziert wird. Diese Klasse wird von Eclipse bei der Aktivierung des Plug-ins geladen und muss von `AbstractUIPlugin` abgeleitet sein.

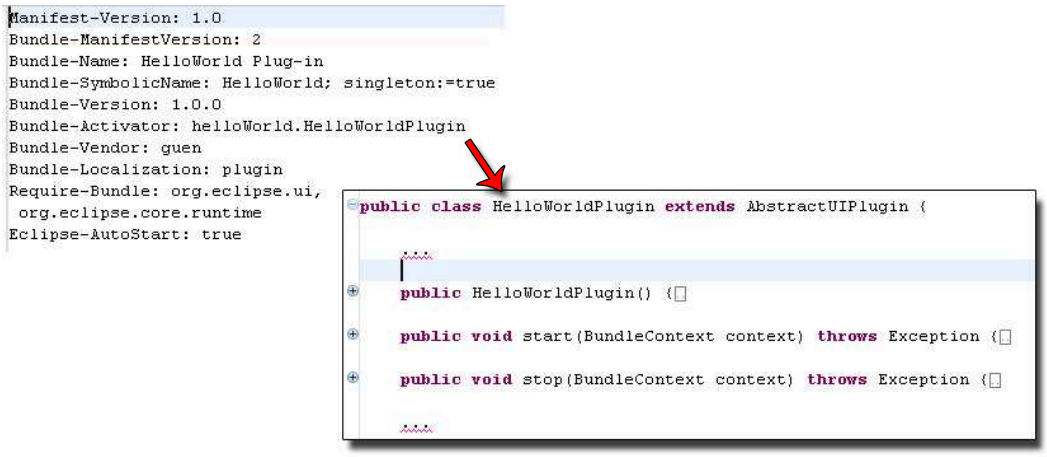


Abbildung 2.5: Eine Manifest-Datei eines Plug-ins und dessen Java-Klasse

2.2.2 Ressourcen-Framework

Das Ressourcen-Framework bildet die Schnittstelle zu dem in Eclipse benutzten *Workspace*. Dabei gibt es zwei Ansichten des *Workspace*. Die physikalische Ansicht, welche die Ordner und Dateien auf der Festplatte bezeichnet, und die logische Ansicht, die die Abbildung des *Workspaces* in Eclipse durch das Ressourcen-Framework darstellt. Das Ressourcen-Framework hat dabei einen viel größeren Umfang und eine größere Funktionalität als die physikalische Sicht. Neben dem Erzeugen, Verwalten oder Löschen von Ressourcen unterstützt dieses Framework auch die Ressourcenereignisverarbeitung sowie Ressourceneigenschaften, *Natures*, *Builder* und Markierungen. Aufgrund dieser zwei Ansichten spielt die Synchronisation beider eine wichtige Rolle. Wenn sich in der physikalischen Ansicht außerhalb der Benutzung von Eclipse etwas verändert, zum Beispiel das Löschen einer Datei, bekommt das Ressourcen-Framework ohne Synchronisation die Veränderung nicht mitgeteilt.

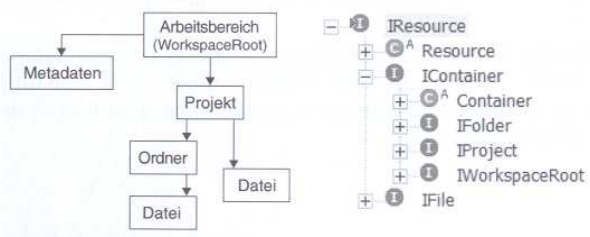


Abbildung 2.6: Schnittstellen der Resourcen [Sha04]

Die Abbildung 2.6 zeigt eine logische Ansicht (links) sowie eine Schnittstellenhierarchie (rechts). Die logische Ansicht wird anhand der physikalischen Ansicht erstellt. Durch verschiedene Schnittstellen des Ressourcen-Frameworks wird ein Zugriff auf diese gewährleistet. So gibt es zum Beispiel eine Schnittstelle `IFile` für den Zugriff auf Dateien oder die verallgemeinerte Schnittstelle `IResource` für alle Dateisystemressourcen.

Nature ist ein Konzept, mit dem das Verhalten und die Funktionen von Eclipse-Projekten festgelegt werden können. Es gibt beispielsweise eine Java-*Nature*, die bestimmte Projektkonfigurationen festlegt, wobei ein Projekt auch mehrere *Natures* haben kann [Sha04].

Builder sind Mechanismen, die vom Arbeitsbereich verwendet werden, um automatisch Dateien zu der Plug-in-spezifischen Logik zu verarbeiten. Dabei werden die Dateien meist in eine andere Form gebracht. Die Kompilierung der Java-Dateien wird zum Beispiel durch einen solchen Builder übernommen [Sha04].

Markierungen sind abstrakte Entitäten zu einer Ressource. Markierungen sind dabei Ressourcenobjekte, die mit anderen Ressourcen verknüpft werden können. Es kann auch innerhalb einer Ressource verknüpft werden. Die Verknüpfung zu einer Zeile innerhalb einer Textressource ist so eine Markierung [Sha04].

2.2.3 Editoren und Ansichten

Die Grundbausteine der Eclipse-Benutzeroberfläche sind Editoren und Ansichten. Hierbei können Editoren und Ansichten von bereits vorhandenen abgeleitet bzw. erweitert sowie neu erstellt werden.
Die Aufgabe von Editoren liegt in der Modifizierung von Dateien bzw. Daten. D.h., ein Editor ist nicht zwingend ein Texteditor und hat somit kein bestimmtes Erscheinungsbild. Daher lassen sich in Eclipse auch jegliche Arten von Editoren einbetten [Sha04]. Das Editor-Framework stellt einige allgemeine Verhaltensweisen von Editoren zur Ableitung zur Verfügung. In Bezug auf Texteditoren werden unter anderem Schnittstellen für Syntax-Highlighting, Inhaltsformatierungen oder Inhaltsassistenten angeboten. Die einzige Vorgabe, die Eclipse in Bezug auf Editoren macht, ist, dass alle Editoren in einem Editorbereich dargestellt werden müssen. Aufgrund dessen werden Ansichten und Editoren auch nicht in ein und demselben Bereich angezeigt.
Unter Ansichten für Eclipse wird alles gefasst, was keine Editoren oder Dialogfelder sind. So gibt es zum Beispiel Ansichten für Ressourcen, Markierungen oder Statusmeldungen. Wie auch bei den Editoren stellt Eclipse ein Framework zur Verfügung, welches die Entwicklung von Ansichten unterstützt. Dabei schlägt Eclipse folgendes Architekturmodel vor:
Es gibt eine strikte Trennung der Daten (*Model*) von der Ansicht (*View*). Die Ansicht `ViewPart` wird von dem Interface `IViewPart` abgeleitet und von der *Workbench* initialisiert. Die Ansicht enthält *Widgets*, wie eine Tabelle oder einen Baum, die die Daten darstellen. Der *Widget* kennt das *Model*, welches die Daten enthält. Jedoch kennt das *Model* nicht den *Widget*. Damit der *Widget* die Daten anzeigen kann, bildet der `ContentProvider` die Schnittstelle zwischen diesen, d.h., dieser definiert, wie der Inhalt aus dem *Model* für die *View* erzeugt wird. Durch den `ContentProvider` werden die Daten auf den *Widget* zwar „zugeschnitten“, für die korrekte Darstellung wird aber zusätzlich noch ein `LabelProvider` benötigt, der für eine genaue Darstellung jedes Datums verantwortlich ist. Dieser gibt in der Regel eine Zeichenfolge oder Symboldarstellung für das entsprechende Datum an den *Widget* zurück.

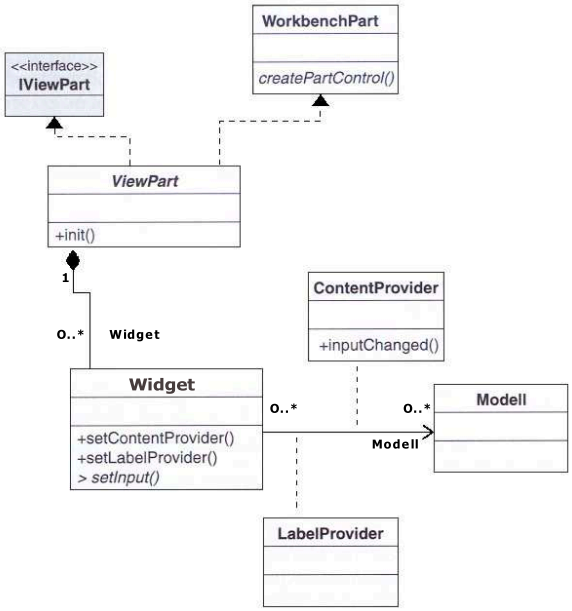


Abbildung 2.7: Architektur einer Ansicht [Sha04]

2.2.4 Perspektiven

Eine Perspektive vereint Ansichten mit Editoren und ermöglicht es aus diesem Grund, die Benutzeroberfläche für Eclipse auf bestimmte Benutzeraufgaben und -rollen auszulegen. Die komplette Ausrichtung und Aufteilung der Oberfläche wird durch eine Perspektive wiedergegeben. Es gibt beispielsweise eine Java-Perspektive, die speziell für die Entwicklung von Java-Programmen oder eine Java-Debug-Perspektive, die optimal für den Debug-Vorgang ausgerichtet ist. Eine Perspektive kann sowohl über die Benutzeroberfläche von Eclipse selbst, als auch über die Plug-in-Entwicklung erstellt werden. Über die Oberfläche kann dies über den Menüpunkt *Window* eingestellt und erstellt werden. Bei der Plug-in-Entwicklung wird die Perspektive durch eine *extension* auf den *extension point* `org.eclipse.ui.perspectives` definiert und durch die Implementierung der Klasse `IPerspectiveFactory` initialisiert. Diese bestimmt lediglich die einzelnen anzuzeigenden Komponenten und richtet diese aus.

2.3 Features - Plug-ins verpacken

Features dienen dazu, Plug-ins zu verwalten, zusammenzufassen und diesen ein *Product Branding* aufzusetzen. Dadurch ergibt sich die Möglichkeit, Plug-ins auszuliefern und diese komfortabel über den *Update Manager* in Eclipse zu installieren. Erst dann ist es Eclipse möglich, die Plug-ins nicht nur auszuführen, sondern auch zu verwalten. Ein *Feature* liefert somit die Steuerung für die Verteilung und Wartung des Plug-in-Codes. Dabei können *Features* mehrere Plug-ins enthalten, auf Plug-ins referenzieren sowie auf anderen *Features* aufbauen (vergleiche Abbildung 2.8). Aufgrund dessen kann beispielsweise sichergestellt werden, dass *Features* als eine Gruppe installiert und konfiguriert werden. Ein *Feature* wird durch ein *Feature*-Manifest in Form einer XML-Datei

definiert [Sha04].

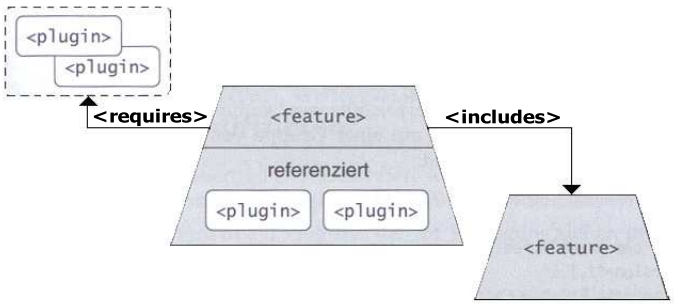


Abbildung 2.8: Features [Sha04]

2.4 Erweiterung durch Plug-ins - GEF

GEF (*Graphical Editing Framework*) ist ein Plug-in, das ein Framework für die Erstellung von grafischen Editoren liefert. GEF wird als *Feature* ausgeliefert und umfasst viele Plug-ins. GEF setzt eine Model-View-Controller-Architektur um und ist spezialisiert auf die Bearbeitung von grafischen Objekten. Die *View*-Schicht der Architektur wird vollständig durch die Bibliothek `Draw2d` realisiert. `Drawd2d` ist sowohl von GEF als auch von Eclipse unabhängig und kann deshalb auch jenseits der Eclipse-IDE verwendet werden. Es baut lediglich auf `SWT` auf, das ebenfalls außerhalb von Eclipse benutzt werden kann. `Draw2d` ist eine mächtige Bibliothek zum Zeichnen von Diagrammen bzw. von so genannten `Figures`, die ein Diagramm ergeben, und `Connections`, welche die Verbindungen zwischen `Figures` darstellen. Es beinhaltet beispielsweise einen mächtigen Layout-Algorithmus für gerichtete Graphen sowie einen *Shortest Path Connection*-Router [EFI06c].

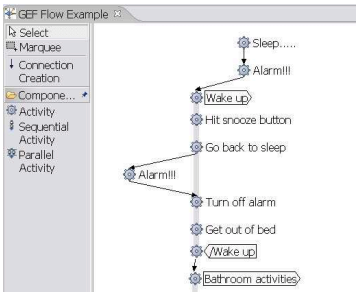


Abbildung 2.9: Beispiel für einen grafischen Editor mit GEF

Die GEF-Architektur bildet eine MVC-Architektur ab (siehe Abbildung 2.10). Das *Model* ist unabhängig von der *View* und der *Controller* verbindet *View* und *Model* miteinander. GEF lässt hier nahezu jegliche Modelle zu. Die Daten aus dem *Model* werden von der *View* durch `Figures` dargestellt, welche durch so genannte `EditParts`, die den *Controller*-Teil übernehmen, mit dem *Model* verbunden werden. Wenn Änderungen am *Model* auftreten, werden die `EditParts` benachrichtigt und können dementsprechend

reagieren. Ebenso geschieht dieses mit Benutzereingaben auf der Oberfläche bzw. an den `Figures`. Die `EditParts` behandeln die *Events* allerdings nicht selbst, sondern haben für jedes *Event* entsprechende `EditPolicies`, die diese Aufgabe übernehmen. Die `EditPolicies` verarbeiten allerdings nur die *Events*. Aktionen führen sie daran anschließend nicht aus. Die Aktionen, wie Änderungen am *Model*, werden von so genannten `Commands` übernommen, die von den `EditPolicies` erstellt werden. Das hat den Vorteil einer erhöhten Modularität, da die Behandlung für jedes *Event* separat implementiert werden kann und die `Figure` unabhängig von ihren Daten sind bzw. jede `Figure` ihren eigenen `EditPart` besitzt [EFI05a].

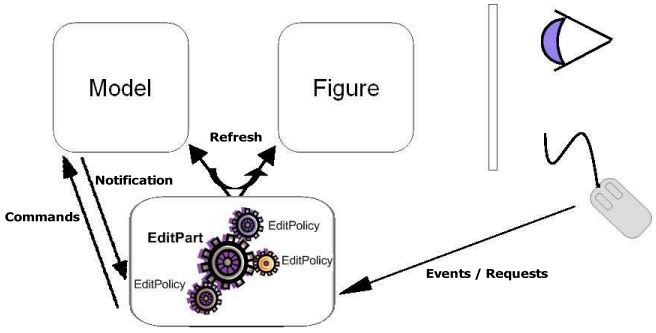


Abbildung 2.10: GEF-Architektur

2.5 Fazit

Eclipse bietet nicht nur eine sehr gute und ausgereifte Architektur für die Plug-in-Entwicklung, sondern auch eine überaus komfortable Entwicklungs-IDE für Plug-ins. Dabei kann auf viele allgemeine Funktionen zurückgegriffen werden, so dass sich die Entwicklung des Plug-ins fast ausschließlich auf die Logik des Plug-ins an sich beschränkt. Diese Tatsache hat den Vorteil, dass in einer bestehenden soliden Architektur entwickelt wird und Überlegungen zum Grundaufbau der Architektur des Plug-ins nicht mehr erforderlich sind. Trotzdem ist Eclipse funktional so offen, dass keine Grenzen gesetzt werden. Dafür spricht auch die Fülle an verwendbaren Plug-ins. Das Plug-in GEF, welches zur Erstellung von grafischen Editoren dient, kann an dieser Stelle als gutes Beispiel angeführt werden. GEF ermöglicht durch eine mächtige grafische Komponente fast jegliche Art der Figuren-Erstellung und die Figuren können mit fast jeglichen Datenmodell verbunden werden. Es soll an dieser Stelle jedoch auch erwähnt werden, dass für den sicheren Umgang eine längere Einarbeitungszeit sowohl in Eclipse selbst als IDE als auch für den Eclipse Plug-in-Mechanismus veranschlagt werden muss. Als Hilfestellung dient die große Eclipse-Gemeinde bzw. die dadurch resultierende ausführliche Dokumentation. Bei Eclipse-Plug-ins, die nicht aus dem Eclipse-Projekt hervorgehen, hängt diese allerdings von der Popularität des Plug-ins ab. Bezüglich der Einarbeitungszeit ist Eclipse in Verbindung mit GEF einer Neuentwicklung eines grafischen Editors meines Erachtens trotzdem vorzuziehen.

Kapitel 3

Fujaba

Niels Hapke

3.1 Einleitung

Im vorliegenden Dokument werden FUJABA und FUJABA für Eclipse (FUJABA4Eclipse) vorgestellt, die zugrunde liegende Architektur erläutert und die Verwertbarkeit für Syspect geprüft.

FUJABA ist ein Computerprogramm, welches mit Hilfe der *Unified Modeling Language* (UML) den Entwurf von Software am Computer erlaubt. Es wurde 1997 ins Leben gerufen und wird derzeitig international, allerdings hauptsächlich in Paderborn und Kassel weiterentwickelt. Der Name FUJABA selbst ist ein Akronym und steht für „From UML to Java And Back again“. Das Programm ist in Java geschrieben und somit plattformübergreifend einsetzbar.

Die Applikation beinhaltet in der Standardversion die UML-Diagrammtypen Klassendiagramm, Zustandsdiagramm und Storydiagramm. Mit Hilfe von FUJABA ist es möglich, aus solchen Diagrammen kompilierbaren und lauffähigen Quellcode zu erzeugen. Aber auch schon vorhandener Quellcode lässt sich in FUJABA importieren und per *Reverse Engineering* als UML Diagramm darstellen [Sch05].
Desweiteren bringt FUJABA ein *Dynamic Object Browsing System* (DOBS) mit sich, welches die interne Objektstruktur in einer *Java Virtual Machine*(JVM) verdeutlicht. So werden beispielweise das Entstehen und Zerstören von Assoziationen und das Verändern von Attributwerten sichtbar gemacht. DOBS wurde entwickelt, um in Schulen die Lehre objekt-orientierter Programmierung zu unterstützen.
Mit im Programm enthalten ist außerdem eine *GUI Bibliothek F Grafik*, welche das problemlose Erstellen von Eingabemasken und Fenstern erlaubt. Somit lässt sich leicht eine Benutzeroberfläche kreieren, die auch Interaktion mit dem Benutzer erlaubt.

Neben FUJABA gibt es auch noch weitere Projekte der *FUJABA Developer Group*,

welche auch auf FUJABA aufsetzen. So beispielsweise die *FUJABA Difference Tool Suite*, welche den Vergleich zwischen zwei UML-Diagrammen erlaubt, oder die *Fujaba Real-Time Tool Suite*, mit deren Hilfe die Spezifikation von eingebetteten Real-Zeit-Systemen mit Komponentendiagrammen und Echt-Zeit-Zustandsdiagrammen möglich ist.

Seit kurzem gibt es nun auch FUJABA für Eclipse. Dieses Plug-in für die Entwicklungsumgebung Eclipse ist bei weitem in der Entwicklung nicht so fortgeschritten wie die *FUJABA Tool Suite*. Es bietet lediglich die Möglichkeit, Klassendiagramme und Storydiagramme zu erstellen. Eine Codegenerierung ist derzeitig noch nicht möglich.

3.2 Architektur von FUJABA

An eine Applikation wie FUJABA und damit auch an die Architektur ergeben sich folgende Anforderungen:

- Die grafischen Diagramme sollen Instanzen von einem Modell, wie zum Beispiel UML oder in unserem Falle einem eigenen UML Profil sein.
- Die Struktur und die Werte der Instanzen sollen durch den Benutzer verändert werden können.
- Das Verändern dieser Instanzen soll wenig rechenintensiv sein.
- Die Darstellung und das Modell müssen konsistent bleiben.

All diese Anforderungen erfüllt das Modell-View-Controller-Konzept, welches in Abbildung 3.1 dargestellt ist. Beim MVC-Modell handelt es sich um ein Architektur-

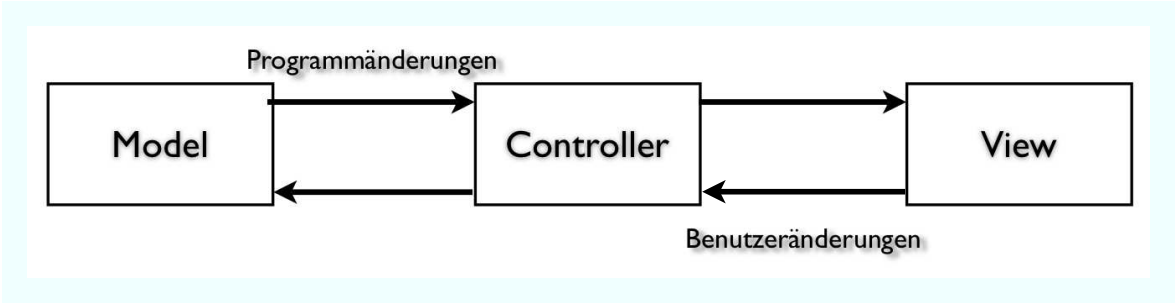


Abbildung 3.1: Das Model-View-Controller Konzept

muster zur Trennung einer Applikation in die drei Sektionen Datenmodell (*Model*), Darstellung (*View*) und Programmsteuerung (*Controller*). Änderungen im *Model* oder der *View* werden über den *Controller* an die jeweilige andere Einheit weitergeleitet. Ziel der Architektur ist die einfache Wiederverwendung einer Sektion beispielsweise des Modells.

Die *Fujaba Developer Group* verfeinert dieses Modell und fügt dem Modell noch ein sogenanntes *Unparse Modul* hinzu. Dieses *Unparse Modul* ergibt mit den Meta-Modell-Instanzen das *Model*. Der *Updater* und der *Fujaba Swing Adapter* (FSA) erfüllen die Aufgaben des *Controllers*. Mit Swing wird die Darstellungssektion implementiert. Zu sehen ist dies in Abbildung 3.2. Bei der erstmaligen Erstellung einer Klasse wird diese

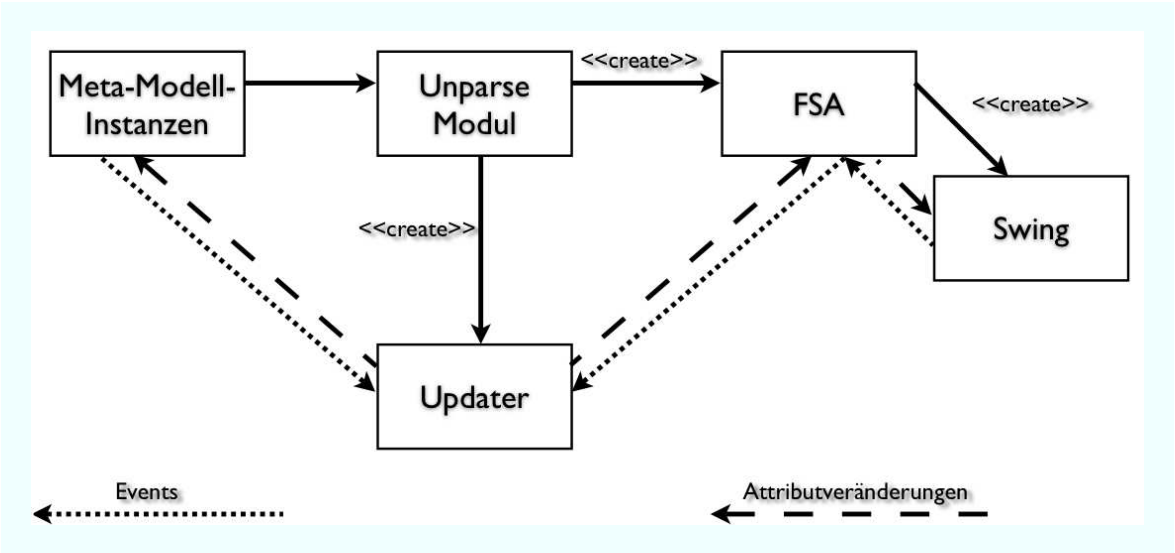


Abbildung 3.2: Die Implementierung des MVC-Modells in FUJABA [Tic05]

in einem entsprechenden Meta-Modell abgebildet, welches in Abbildung 3.3 als Beispiel zu sehen ist. Jedes Element in der oben abgebildeten UML-Klasse (Klasse, Attribut, Attributtyp) hat eine eigene Instanz. Beim Erzeugen eines weiteren Attributes würde eine weitere Instanz von der Klasse `UMLAttr` erzeugt werden, welche eine Assoziation zur bestehenden u0:UMLCLass und zu einer weiteren Instanz von `UMLBaseTypes` hätte.

Jeder Klasse im Meta-Modell gehört ein *Unparse Modul* an. Dieses wird im Falle einer Instanziierung der Klasse aktiv und erzeugt den zugehörigen *FUJABA Swing Adapter* und einen *Updater*.
Der FSA übernimmt die Aufgabe des *Controllers*. Änderungen an der Darstellungsschicht oder der Meta-Modell-Instanzen werden über ihn abgewickelt. Realisiert wird dies über das bekannte Observer-Pattern. Die Meta-Modell-Instanzen und die Swing-Komponenten implementieren sogenannte *PropertyChangeClients*, welche bei Veränderungen durch das Programm oder den Benutzer *PropertyChangeEvents* „feuern“. Diese werden durch eine Implementierung des *PropertyChangeListeners* im FSA und *Updater* aufgenommen und entsprechende Maßnahmen werden eingeleitet.
Der *Fujaba Swing Adapter* selbst kapselt außerdem verschiedene Swing-Elemente. So kann die Instanz von `FSACombobox`, welche unter anderem bei Attributen kennzeichnet, ob es sich um ein öffentlich zugängliches, privates oder gesichertes Attribut handelt, mehrere Ausprägungen haben. Nämlich das Icon für `public`, `private`, `secure` oder `packaged`.

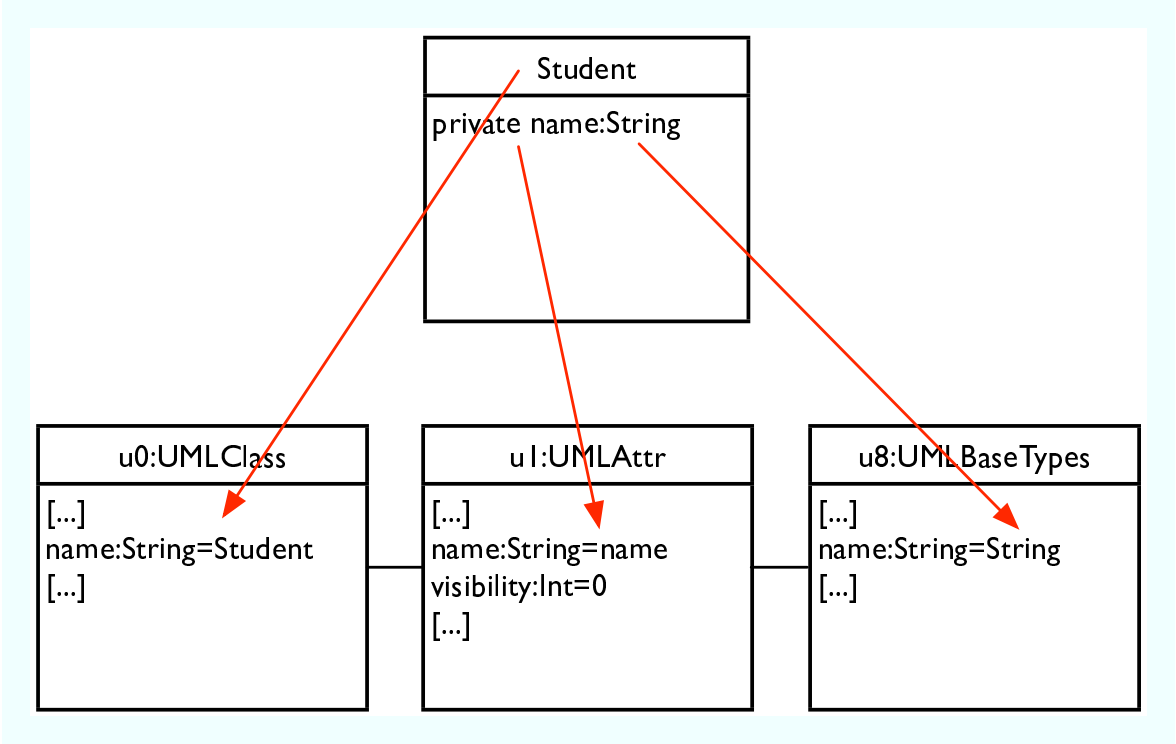


Abbildung 3.3: Das FUJABA zugrunde liegende Meta-Modell [Tic05]

Der bereits erwähnte *Updater* soll nun näher erläutert werden. Dieser hat neben der Rolle des Observers im gleichnamigen Modell auch noch die Möglichkeit, Logik in FSA Eigenschaften zu übersetzen und umgekehrt FSA Eigenschaften in Logik. Dies ist zum Beispiel im *Dynamic Object Browsing System* von Bedeutung. So kann der Wert einer bool'schen Variable in Farben übersetzt werden. Die Variable mit der Auspägung "false" würde übersetzt "Rot" und "true" würde "Grün" ergeben.

3.3 Architektur von FUJABA4Eclipse

In der FUJABA-Version für Eclipse ist eine teilweise andere Struktur vorhanden. Zugrunde liegt die durch das Graphical Editing Framework (einem Eclipse Plugin) vorgegebene Struktur, welche auf dem MVC-Modell aufsetzt. Siehe hierzu Abbildung 3.4. Das Zusammenspiel der Komponenten ist am Besten mit Hilfe eines Beispiels zu erklären. Wir gehen davon aus, dass in unserem Klassendiagramm zwei Klassen existieren. Diese wollen wir durch eine Assoziation verknüpfen.
Durch Anwählen des Icons zum Erstellen einer Assoziation wird ein Befehl auf den Befehlsstapel des Editors gelegt. Bei der Bearbeitung dieses Befehls wird die Instanz `EditPart`, welche die Rolle des *Controllers* zusammen mit den Instanzen `EditPolicy` und `Command` übernimmt, kontaktiert. Diese erzeugt das Fenster zum Erstellen und Bearbeiten einer Assoziation. Nach Auswahl der Klassen, welche miteinander verknüpft werden sollen und anschließendem Bestätigen des Dialogs, wird der erzeugte Befehl an

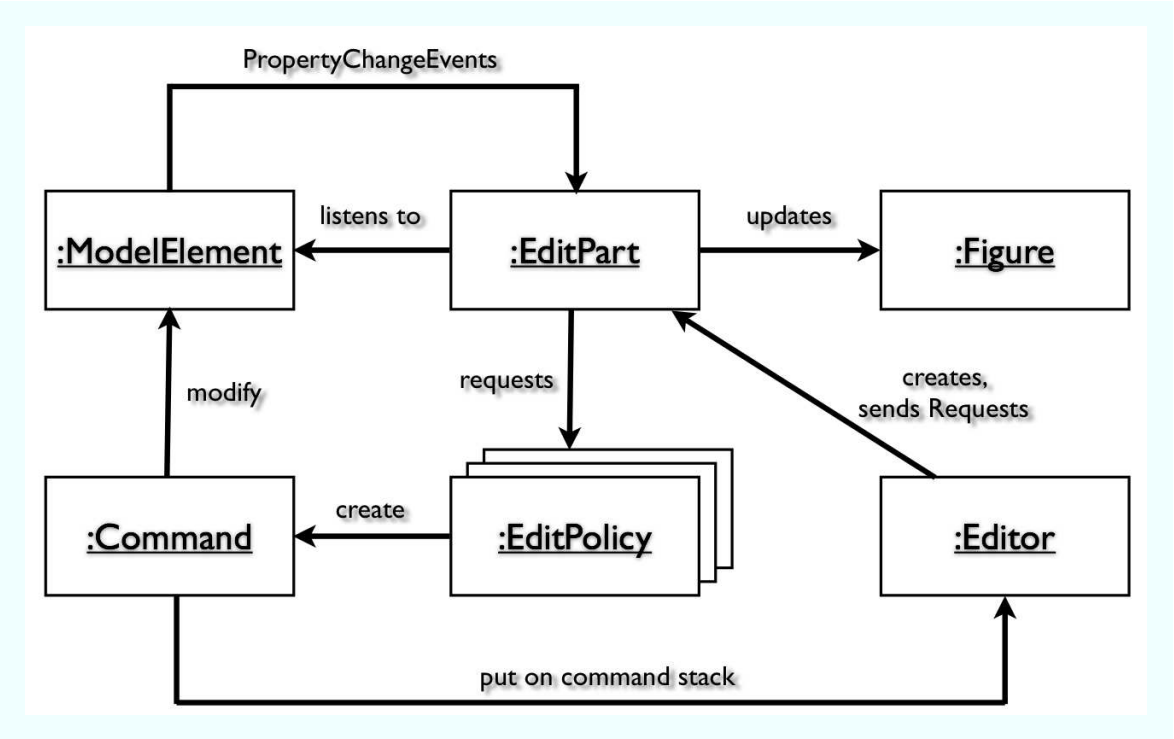


Abbildung 3.4: Die FUJABA4Eclipse Struktur [Mey05a]

die `EditPolicy` übergeben. Diese überprüft unter anderem, ob überhaupt eine Assoziation zwischen den beiden Klassen möglich ist. Ist dies möglich, wird erst der eigentliche Befehl zur Veränderung der *Model*-Elemente erzeugt. Bei der Erzeugung der Assoziation im Meta-Modell wird, wie auch schon in FUJABA, ein `PropertyChangeEvent` „gefeuert“. Der als *Observer* dienende `EditPart` führt dann die Veränderungen in der Darstellung, den *Figures*, durch.

3.4 Zusammenfassung mit Hinblick auf eine Weiterverwendung im Rahmen des Projektes

Die Konzepte, welche FUJABA und auch FUJABA4Eclipse zugrunde liegen, sind durchaus für eine Weiterverwendung geeignet. Die Struktur ist klar und übersichtlich vorgegeben. Durch den in FUJABA eingeführten FSA wird außerdem eine Abstraktion von der GUI-Implementierung erreicht. Das Meta-Modell und das *Unparse Modul* machen den Eindruck, auch bei Erweiterungen durch ihren modularen Aufbau keine Schwierigkeiten mit sich zu bringen.

Weniger gut stellen sich die Erweiterungmöglichkeiten der bestehenden Applikationen dar. Während FUJABA4Eclipse noch, verglichen mit FUJABA, recht übersichtlich wirkt, erschwert doch die unzureichende Dokumentation eine Erweiterung sichtlich. Lediglich die Oberpakete sind mit Kommentaren versehen. Einzelne Klassen

besitzen dagegen lediglich den durch Eclipse vergebenen Standardkommentar *"No comment provided by the developer, please add a comment by yourself to improve documentation"*[FDG05]. Dieser Missstand ist den Entwicklern durchaus bekannt. Verbesserungen existieren leider nur in neueren Projekten. Die Möglichkeit einer persönlichen Einführung in die Entwicklung durch einen Besuch in Kassel bei Albert Zündorf entschädigt dafür nur teilweise.

FUJABA und FUJABA4Eclipse bieten außerdem die Möglichkeit der Erweiterung durch Plug-ins. Diese Alternative ist die einzige sinnvolle und mögliche Erweiterung. Für beide Applikationen sind jeweils ein PDF-Dokument eines Vortrages vorhanden ([Wen05] und [Mey05a]). Bei der Erweiterung von FUJABA4Eclipse ist zu beachten, dass dieses sehr instabil ist und derzeitig nur unter Eclipse 3.02 funktioniert.
Das Nutzen einzelner Klassen gestaltet sich durch oben genannten Missstand ebenso schwierig. In Frage würden die Klassen des Meta-Modells der Klassen zum Zeichnen der Diagramme kommen.

Abschließend lässt sich also sagen, dass eigentlich nur eine Erweiterung von FUJABA durch ein Plug-in in Frage kommt. Jede andere Erweiterung bringt einen zu großen Einarbeitungsaufwand mit sich. FUJABA4Eclipse ist derzeitig noch im Alpha-Stadium und dadurch kaum nutzbar.

Kapitel 4

ArgoUML

Patrick Kuballa

4.1 Einleitung

Die Projektgruppe Syspect hat das Ziel, eine grafische Entwicklungsumgebung zu entwerfen und zu implementieren, welche eine Unterstützung für eine formale UML-Teilsprache bietet. In der Abteilung "Entwicklung korrekter Systeme" der Universität Oldenburg wurde eine Spezifikationssprache namens CSP-OZ-DC entwickelt, die die Aspekte Daten, Zeit und Reaktion beschreiben kann. Diese Spezifikationssprache soll bekannte UML-Diagramme ergänzen, um Wechselwirkungen unter den Diagrammen zu beschreiben. In einem weiteren Schritt kann dann das modellierte System mittels einem Model-Checker auf Korrektheit überprüft werden. ArgoUML ist ein grafisches Entwicklungswerkzeug, mit dessen Hilfe sich UML-Diagramme zeichnen lassen. Diese Ausarbeitung gibt einen kurzen Überblick über das Open-Source-Projekt ArgoUML und dessen Funktionalität. Ziel ist die Prüfung der Software auf eine Erweiterbarkeit zur Umsetzung der Anforderungen, die aus der Projektbeschreibung hervorgehen.

4.2 Allgemeines zu ArgoUML

In diesem Kapitel sind allgemeine Informationen zum Open-Source-Projekt ArgoUML enthalten. Es soll dem Leser einen kurzen Überblick über das Projekt selbst, dessen Funktionalität, besondere Eigenschaften und Hinweise zur Dokumentation geben.

Was bedeutet der Name ArgoUML: Der Name des Projekts ArgoUML setzt sich aus den zwei Wörtern "Argo" und "UML" zusammen.

Die Unified Modeling Language (vereinheitlichte Modellierungssprache), häufig mit UML abgekürzt, ist eine von der Object Management Group OMGs entwickelte und standardisierte Beschreibungssprache, um Strukturen und Abläufe in objektorientierten Softwaresystemen darzustellen [Wik05b].

Der Begriff “Argo” stammt vom Entwickler Jason Roberts der Bibliothek für kognitive Unterstützung.

Def. Kognition: Mit dem Begriff Kognition werden solche Prozesse und Produkte bezeichnet, die auf der Grundlage der Leistungsfähigkeit des Gehirns auf überwiegend intellektuelle, verstandesmäßige Wahrnehmungen und Erkenntnisse bezogen sind [Wik05d].

Der Begriff im Kontext der Bibliothek Argo steht hier für Erkenntnis- und Entscheidungsprozesse, mit denen Anwender von Programmen, die diese Bibliothek nutzen, in ihrer Tätigkeit unterstützt werden. Jason Roberts war seinerzeit auf der Suche nach einem passenden Namen in der griechischen Mythologie, und wurde in der Geschichte “Jason and the Argonauts” fündig. Er benannte seine Bibliothek nach dem Schiff aus der Geschichte, welches hellseherische Fähigkeiten besaß. Argo ist zudem ein Teil von Programmnamen geworden, die diese Bibliothek nutzen.

Allgemeine Informationen zum Projekt ArgoUML: ArgoUML ist ein Open-Source-Projekt, welches unter ständiger Erweiterung steht. Die erste Version wurde im Jahre 1996 veröffentlicht und steht heute unter BSD-Lizenz zum freien Download unter [tig05] zur Verfügung.

Auszug BSD Lizenz: ...Bei der BSD-Lizenz handelt es sich um ein standardisiertes Copyright für Freie Softwareprodukte. Software unter der BSD-Lizenz darf frei verwendet werden und es ist erlaubt, sie zu kopieren, zu verändern und zu Verbreiten [Wik05c].

Eine kommerzielle Erweiterung bietet z.B. das Unternehmen Gentleware mit dem Produkt “Poseidon” an.

4.2.1 Benutzeroberfläche

Abbildung 4.1 zeigt die Benutzeroberfläche von ArgoUML. Zu sehen ist die Menüleiste, auf der in *Drop Down*-Menüs sämtliche Funktionen zugreifbar sind. Darunter befindet sich eine Werkzeugleiste mit weiteren ausgewählten Funktionen. In der Mitte rechts ist das Editorfenster zum Zeichnen der Diagramme zu sehen, links daneben das Navigator-Panel, mit dessen Hilfe die Struktur der Diagramme aufgezeigt wird. Unten rechts ist das Eigenschaften-Panel untergebracht, hier werden Eigenschaften von aktiven Elementen angezeigt und können editiert werden. Links daneben befindet sich die sogenannte „To-Do“- Liste, auf der Aufgaben anzeigt werden. Des weiteren soll hier nicht weiter auf die Benutzung der Oberfläche und Anordnung der Elemente eingegangen werden, da dieses den Umfang der Ausarbeitung sprengen würde, somit verweise ich hier auf das online bereitgestellte Benutzerhandbuch, in dem sämtliche Elemente detailliert erklärt werden.

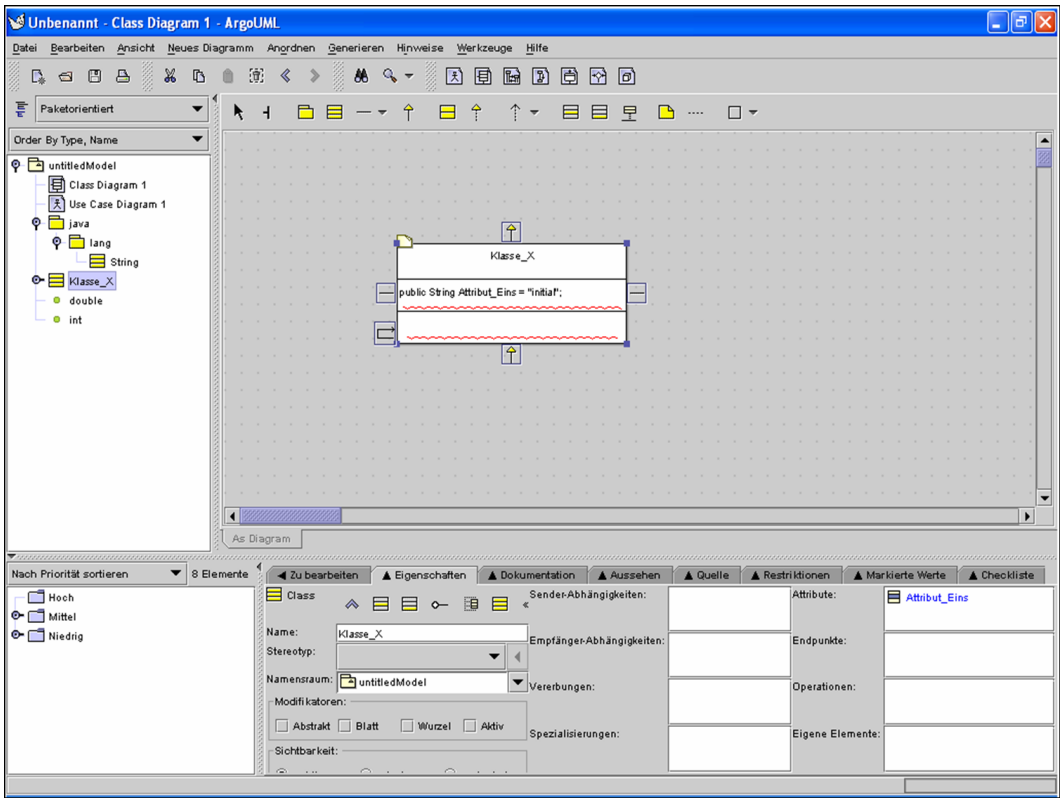


Abbildung 4.1: Benutzeroberfläche ArgoUML

4.2.2 Funktionen von ArgoUML

In diesem Kapitel werden die einzelne Funktionen erläutert, die ArgoUML dem Anwender bietet.

Grafische Oberfläche: ArgoUML bietet dem Anwender eine grafische Oberfläche (siehe Abbildung 4.1), mit der es möglich ist, alle zur Verfügung stehenden Diagrammarten zu zeichnen. Die einzelnen Fenster auf der Oberfläche lassen sich nach individuellen anordnen, ein- und ausblenden. Die folgenden Diagrammarten werden in der Version 0.19 unterstützt.

- Klassendiagramm
- Use-Case Diagramm
- State machines
- Activity Diagramm
- Collaboration Diagramm
- Deployment Diagramm

Forward Engineering: *Forward Engineering* beschreibt die Möglichkeit, erstellte Diagramme in einen Quellcode einer definierbaren Zielsprache zu übersetzen. Bis dato wurde die Zielsprache Java fest in ArgoUML integriert. Weitere Sprachen sind als Module vorhanden und können bei Bedarf nachinstalliert werden.

Reverse Engineering: *Reverse Engineering* beschreibt den Prozess, aus bereits verfügbaren Quellcode Diagramme zu erstellen. Zur Zeit wird diese Funktion nur für die Zielsprache Java angeboten.

Object Constraint Language Support: Für UML-Klassendiagramme besteht die Möglichkeit, mit dem *Object Constraint Language Support* eingegebene Daten auf Syntax und Typen zu überprüfen.

XMI-Support: *Extensible Metadata Interchange* (XMI) ist ein Datenformat, welches zum Austausch von Daten unter Programmen dient. Bei ArgoUML wird es dazu benutzt, ein entwickeltes Modell in diesem Datenformat zu speichern, um es anderen Programmen zur Verfügung zu stellen.

Export von Diagrammen: In ArgoUML ist es möglich, Diagramme in ein Grafikformat zu exportieren. Mögliche Dateiformate sind GIF, PGML, SVG, PostScript und Encapsulated PS.

Internationalisierung: In ArgoUML besteht die Möglichkeit, sämtliche Bedienelemente wie u.a. Menüeinträge in verschiedenen Sprachen wie Deutsch, Englisch etc. darzustellen.

Kognitive Unterstützung: Der Entwickler wird beim Modellieren seines Systems in ArgoUML durch verschiedene Funktionen unterstützt. Hierauf wird in Kapitel 4.2.3 näher eingegangen.

4.2.3 Kognitive Unterstützung:

Design-Critics *Design-Critics* sind Agenten die im Hintergrund von ArgoUML ablaufen und für den Anwender nicht sichtbar sind, da sie den Tätigkeitsablauf des Entwicklers nicht unterbrechen sollen. Der Sinn der Agenten besteht darin, den Verlauf der Modellierung, sowie Design-Aspekte des Modells während der Entwicklung zu überwachen und ggf. den Anwender in Form von Anmerkungen bzw. Verbesserungsvorschlägen zu informieren. Dieses geschieht durch Einträge des Agenten in die so genannten ToDo-Listen, auf die später noch eingegangen wird. Zu den Anmerkungen des Agenten gehören Designentscheidungen wie z.B. die geeignete Wahl für Bezeichner von Klassen, sowie Hinweise zu inkonsistenten Diagrammen.

Corrective Automations: Dieses bezeichnet eine Funktion, mit dem der Anwender aus einem Element der To-Do Liste einen automatischen Verbesserungsvorschlag durchführen kann. Hierzu zählen z.B. die Umbenennung von Bezeichnern.

Verschiedene Sichten: In ArgoUML hat der Anwender die Möglichkeit, zwischen verschiedenen Ansichten eines Diagramms zu wechseln. Hierdurch soll es dem Benutzer ermöglicht werden, verschiedene Detailstufen eines Diagramms zur Verfügung zu haben, um die Übersicht zu fördern.

Alternative Design-Repräsentation: Die modellierten Diagramme können in verschiedenen Arten angezeigt werden. So kann der Entwickler ebenfalls unterschiedliche Sichtweisen auf die Diagramme erhalten. Möglichkeiten sind eine grafische Darstellung, eine Darstellung als Quellcode, sowie eine tabellarische Anzeigeform, aus denen gewählt werden kann.

ToDo-Listen: ToDo-Listen enthalten Aufgaben, die für den Designprozess wichtig sein könnten. Die Einträge können entweder manuell vom Entwickler oder durch den *Design-Critics*-Agenten erzeugt werden und sind in drei unterschiedliche Prioritätsstufen unterteilt.

Check-Listen: Check-Listen stellen dem Anwender eine Liste zur Verfügung, in der wesentliche Design-Entscheidungen zur aktuellen Diagrammart enthalten sind. Der Anwender soll durch die Listen auf Konventionen und Besonderheiten aufmerksam gemacht werden, um möglichst alle Design-Entscheidungen günstig zu treffen.

4.2.4 Dokumentation von ArgoUML

Im folgenden Kapitel wird kurz auf die vorhandene Dokumentation von ArgoUML eingegangen.

Handbuch, Tutorial, Tour, FAQ: Auf der Projekt-Homepage steht dem Anwender neben einem Benutzerhandbuch zum Umgang mit ArgoUML auch eine Tour, ein Quick-Guide sowie eine FAQ-Seite zur Verfügung.

Cookbook: Für Entwickler die ArgoUML in dessen Funktionalität erweitern möchten, steht ein *cook book* (engl. für Kochbuch) zur Verfügung. In diesem Buch wird die Architektur in allen Einzelheiten dokumentiert, sowie Erläuterungen zum Programmierungen von Modulen und Plug-ins zu ArgoUML gegebenen [Kli05].

ArgoUML-Quellcode-Dokumentation: Der Quellcode ist durch JavaDoc dokumentiert und kann auf der Projekt-Homepage eingesehen werden.

4.3 Projektrelevante Themen

4.3.1 Aufgaben der Projektgruppe Syspect

Die Projektgruppe Syspect hat das Ziel, ein UML-Entwicklungsumgebung zu entwickeln und zu implementieren, welches die im folgenden aufgeführten Merkmale aufweisen soll.

- UML Werzeug um Unterstützung eines CSP-OZ-DC Profils erweitern
- Unterstützung von
  - Klassendiagrammen
  - State Machines
  - Composite Structure Diagrammen
- Formalisierung der Daten durch visuelle Eingabe der Spezifikationssprache Z
- Eingabe der Zeitaspekte durch Formeln des Duration Calculus (DC)
- Speicherung der UML-Modelle im Standardaustauschformat (XMI)
- Export als Latex-Sourcecode
- Export als Phasen-Event-Automaten (PEA)

4.3.2 ArgoUML Intern

Dieses Kapitel beschäftigt sich mit der internen Struktur von ArgoUML. Es soll einen kurzen Einblick über die verwendeten Bibliotheken sowie einen Einblick in Organisation des Quellcode geben. Auf die einzelnen Funktionen von Paketen wird hier nicht näher eingegangen.

Externe Bibliotheken die von ArgoUML verwendet werden:

- Das *Graphical Editing Framework* (GEF) dient zur visuellen Darstellung der einzelnen Diagramme, sowie das Manipulieren dieser Elemente.
- Die *Extended JToolbar* wird von ArgoUML benutzt, um einige Funktionen in Form von Icons zugreifbar zu halten.
- *I18n (Internationalization)*: Dies ist ein Dienst zur Bereitstellung von verschiedenen Sprachen wie Deutsch, Englisch etc.
- NSUML ist eine Implementierung der UML-Metasprache 1.3, die als Grundlage für die verschiedenen Diagrammarten verwendet wird. Hierin sind alle Diagrammarten und deren Beziehungen ausgedrückt.
- *Xerces* dient zum Parsen von XML-Dokumenten, die von ArgoUML eingelesen werden sollen.
- Das *UML Metamodel Interface* (JMI) dient zum Zugriff auf das durch die NSUML-Bibliothek bereitgestellte UML-Metamodell.
- *Log4j* (Logging-Framework) stellt eine Funktionalität zur Verfügung, die es ArgoUML erlaubt, interne Meldungen in einer Log-Datei zu führen.
- ANTLR (Translator Generator) wird von ArgoUML benötigt, um *Reverse Engineering* zu ermöglichen.

Organisation der ArgoUML-Bibliotheken: Der Quellcode von ArgoUML ist in so genannte *subsystems* unterteilt. Die Idee ist, dass jedes *subsystem* nur für einen kleinen Funktionsbereich zuständig ist, um den Quellcode möglichst flexibel und erweiterungsfreundlich zu halten. Jedes *subsystem* besitzt einen eindeutigen Namen und ist in einem eigenen Verzeichnis gespeichert.
Jedes *subsystem* ist wiederum in *Layers* (Schichten) unterteilt. Je tiefer eine Ebene in einem *subsystem* enthalten ist, desto spezieller wird dessen Funktionalität.

Prinzip von ArgoUML: In ArgoUML wird das Model-View-Controller-Prinzip angewendet. Das Modell stellt hierbei die verwendete NSUML-Bibliothek dar, in der alle Eigenschaften, die ein Modell annehmen kann, abgebildet sind. Die *View*-Komponente wird durch die GEF-Bibliothek realisiert, welches die Möglichkeit bietet, eine grafische Oberfläche zum Zeichnen bereitzustellen. In ArgoUML selbst ist die Logik implementiert, die als Schnittstelle zwischen den beiden anderen Komponenten fungiert.

Aufbau eines Diagramms: Eine Diagrammart wie z.B. das Klassendiagramm besteht in ArgoUML aus mehreren Einheiten, die im Quellcode über verschiedene Verzeichnisse verteilt sind. Das *subsystem* `org.model.argouml.uml` enthält die zwei Verzeichnisse `diagram` und `ui`. Das Verzeichnis `ui` enthält in einem weiteren Unterverzeichnis die Spezifikation für das zum Diagramm zugehörige Eigenschaftsfenster. Im Verzeichnis `diagram` sind in weiteren Verzeichnissen die grafischen Elemente der einzelnen Diagrammarten untergebracht, sowie der Zugriff auf das Modell jeder einzelnen Diagrammart implementiert. In einem weiteren Verzeichnis namens `org.argouml.model.uml` finden sich die so genannten *Factory*- und *Helper*-Klassen zum jeweiligen Diagramm wieder. Die *Factories* beinhalten Funktionen zum Generieren bzw. Löschen von Diagrammelementen, die *Helper*-Klassen Methoden zum Manipulieren einzelner Elemente.

4.4 Handlungsbedarf der Projektgruppe:

Die einzelnen Anforderungen, die umgesetzt werden sollen, werden hier auf Umsetzbarkeit durch eine Erweiterung von ArgoUML untersucht.

Unterstützung für Klassendiagramme und State Machines: Beide Diagrammarten sind bereits in ArgoUML implementiert. Da es eine Eingabe von verschiedenen Zeit- und Abhängigkeitswerten geben soll, müssten diese Diagramme um diese Funktionalität erweitert werden. Eine Erweiterung würde die visuellen Elemente sowie das zugehörige Eigenschaftsfenster beinhalten, die es ermöglicht, gerade diese Werte einzugeben und darzustellen.

Unterstützung von Kompositstrukturdiagrammen: Da diese Diagrammart durch ArgoUML selbst nicht unterstützt wird, müsste sie neu implementiert werden. Das Problem besteht darin, dass durch das zur Verfügung gestellte Modell durch die

NSUML-Bibliothek die Spezifikation von UML 1.3 verwendet wird, in der manche Elemente nicht enthalten sind, die für das zu entwickelnde Diagramm benötigt werden. Eine Verfremdung von bereits zur Verfügung gestellten Elementen aus der Spezifikation von UML 1.3 ist generell möglich, um eine neue Diagrammart zu implementieren, es besteht jedoch Grund zu der Annahme, dass z.B. der Export in das Datenformat XMI nicht mehr anwendbar ist, da für verschiedene Diagrammarten dann gleiche Elemente benutzt würden und eine Interpretierung nicht mehr möglich wäre.

Eingabe von Daten der Spezifikationsprache Z: Die Eingabe von Daten der Spezifikationssprache Z würde die visuellen Elemente, das Verhalten und die Eigenschaftsfenster eines Diagramms betreffen. Diese müssten um die entsprechende Funktionalität erweitert werden.

Eingabe der Zeitaspekte durch Formeln des Duration Calculus: Hier ist dieselbe Erweiterung erforderlich, die im vorhergehenden Absatz beschrieben wurde.

Speicherung der UML-Modelle im Standardaustauschformat XMI: Die Grundfunktionalität zur Speicherung im Austauschformat XMI ist in ArgoUML bereits integriert. Zu klären ist, inwieweit eine Erweiterung Einfluss auf die Speicherung nehmen würde. Gegebenenfalls müsste dann die Implementierung der XMI-Funktionen in ArgoUML angepasst werden.

Export als LaTeX-Quellcode: Der Export in LaTeX-Quellcode könnte durch einen in der Menüleiste hinzugefügten Menüeintrag geschehen. Wie eine Konvertierung in die Zielsprache erfolgen könnte, bleibt zunächst ungeklärt.

Export in Phasen-Event-Automaten: Hier ist dieselbe Erweiterung wie auch bei der Eingabe von Daten der Spezifikationsprache Z vorzusehen.

4.5 Fazit

Im Rahmen der Projektgruppe Syspect wurde eine Vortragsserie der einzelnen Teilnehmer gehalten, in denen projektrelevante Themen wie auch ArgoUML erläutert wurden. Anschließend an jeden Vortrag wurde über Fragestellungen diskutiert, um mögliche Probleme oder Bedenken zu äußern.

Das UML-Modellierungstool ArgoUML bietet meiner Meinung nach eine gute Basis für eine Erweiterung zur Umsetzung der gesetzten Anforderungen. Viele Funktionen die das zu entwickelnde Werkzeug bieten soll, sind bereits in ArgoUML implementiert, so dass wichtige Implementierungszeit eingespart werden könnte.

ArgoUML selbst steht unter ständiger Entwicklung, bietet eine umfassende Dokumentation und ist von Grund auf dafür ausgelegt, für Erweiterungen offen zu sein. Desweiteren besteht die Möglichkeit, bei aufkommenden Problemen bei der

Erweiterung des Systems durch unsere Projektgruppe, die aktiven Entwickler von ArgoUML zu kontaktieren, um Hilfe bei Problemstellungen zu erhalten.

Das ArgoUML die Grundlage für ein verwendbares Produkt darstellen kann, zeigt insbesondere, dass es von dem Unternehmen Gentleware mit Ihrem Produkt "Poseidon" ebenfalls verwendet wurde und kommerziell vermarktet wird.

Die Einarbeitungszeit in die Struktur von ArgoUML und die Anpassung an unsere Bedürfnisse darf nicht unterschätzt werden. Im Gegensatz dazu steht jedoch eine völlige Neuentwicklung eines solchen Systems, wodurch ebenfalls ein immenser Zeitaufwand und zusätzliche Probleme auf die Projektgruppe zu kommen können.

Kapitel 5

Community Z Tools

Henning Rohlfs

5.1 Einleitung

Community Z Tools [CZT05] ist ein Open-Source-Projekt, das sich zur Aufgabe gemacht hat, Werkzeuge für den Umgang mit in Z geschriebenen formalen Definitionen zu entwickeln. Außerdem sollen Object-Z sowie *Timed Communicating Object-Z* (TCOZ) unterstützt werden. Als Motivation für die Entwicklung der *Community Z Tools* ist das Fehlen von miteinander zusammenarbeitenden Applikationen zu sehen, die zusammen eine Entwicklungsumgebung für Z darstellen könnten. Dementsprechend ist die Zielsetzung für das Projekt die Entwicklung folgender Werkzeuge:

- XML Schema für Z
- Java Klassen für einen *Z annotated syntax tree* (AST), also eine Darstellung von Z Ausdrücken in Java
- Konvertierungswerkzeuge zwischen XML und dem Java AST
- Grafische Editoren zum Eingeben von Z-Ausdrücken
- Einen Z-Animator
- Exportwerkzeuge zum Übersetzen von Z-Ausdrücken in andere Darstellungsformen wie zum Beispiel LaTeX

Weiterhin sollen diese Werkzeuge die oben genannten Erweiterungen (Object-Z sowie TCOZ) unterstützen.

5.2 Werkzeuge

Aus den in der Einleitung beschriebenen Zielsetzungen ergeben sich folgende Unterprojekte, die zusammen das *Community Z Tools*-Projekt ausmachen:

ZML: Dies ist die XML Darstellung für Z Terme.

Corejava: Hierbei handelt es sich um die Java-Klassen zur Repräsentation des Syntax-Baumes für Z-Ausdrücke. Zusätzlich sind hier bereits das Lesen sowie Schreiben von ZML implementiert. Die Erweiterungen für Object-Z und TCOZ sind noch in Entwicklung.

GnAST: GnAST ist ein Quellcode-Generator, der aus einem XML Schema Java-Klassen und *Interfaces* generiert. Hiermit werden die Corejava-AST-Klassen generiert. Allerdings ist GnAST bisher nur als internes Hilfsmittel für die CZT Entwickler im Einsatz.

JEdit Plugins: Dieses Unterprojekt hat zum Ziel, das Eingeben und Verarbeiten von Z-Ausdrücken zu vereinfachen.

Parser: Der Parser ist dafür zuständig, einen AST aus anderen Darstellungen für Z Ausdrücke zu generieren. Ebenso kann mit dem Parser ein AST in eine alternative Darstellungsform überführt werden. Bisher unterstützt er Z und Object-Z. TCOZ ist bisher nur geplant. Als Darstellungsformen sind Unicode sowie LaTeX möglich. Weiterhin ist die Implementierung eines E-Mail-Markup geplant.

Typechecker: Der Typechecker dient zum Überprüfen von Z-Ausdrücken auf Typkorrektheit.

Gaffe: Gaffe ist ein GUI-Builder für Z-Animatoren.

ZLive: ZLive ist ein Z-Animator, der auf den Ideen von Jaza basiert, einem in Haskell geschriebenen Animator.

ZMLTP: Dies ist ein Projekt mit dem Ziel, HTML aus ZML zu generieren.

Viele der Tools sind noch in einer relativ frühen Entwicklungsphase, wie zum Beispiel ZMLTP.

5.3 ZML

ZML ist ein Datenaustauschformat für Z und soll dazu dienen, Anwendungen verschiedener Entwickler zu integrieren, um insgesamt eine Umgebung zu erhalten, in der Z entwickelt werden kann. Wie der Name schon andeutet, ist ZML als XML realisiert. Es basiert auf einem XML Schema, das im ZML Unterprojekt entwickelt wird. Dieses basiert auf dem ISO-Standard für Z.

5.3.1 Aufbau

Ein Z Term:

$x : \mathbb{N}$

wird als

```
<VarDecl>
   <DeclName>
      <Word>x</Word>
   </DeclName>
   <RefExpr Mixfix="false">
      <RefName>
         <Word>ℕ</Word>
      </RefName>
   </RefExpr>
</VarDecl>
```

dargestellt. Der ZML-Ausdruck hat folgenden Aufbau: Zunächst wird deklariert, dass es sich um eine Variablendeklaration handelt. Der Name der zu deklarierenden Variablen ist x. Im zweiten Teil der Deklaration wird festgelegt, dass X vom Typ $\mathbb{N}$ sein soll. Andere Ausdrücke werden entsprechend umgeformt.

5.4 JEdit-Plug-ins

In diesem Unterprojekt des *Community Z Tools*-Projekts ist eine Sammlung von Werkzeugen und Erweiterungen für den in Java geschriebenen Texteditor JEdit zu finden.

5.4.1 CZTSans

Hierbei handelt es sich um eine Schriftart, in der zusätzliche Symbole für Z eingearbeitet wurden. Somit ermöglicht sie die Darstellung von Z-Termen als Unicode-Text.

5.4.2 ZCharMap

ZCharMap ist eines der JEdit-Plug-ins des *Community Z*-Projekts. Es stellt ein Interface zur Eingabe von Z-Ausdrücken in LaTeX sowie Unicode dar. Außerdem hat sie die Funktion, mit Hilfe des Typecheckers die Ausdrücke zu überprüfen und sie in alle drei bisher implementierten Darstellungen (ZML, Unicode, LaTeX) zu konvertieren.

Technisches

Das JEdit-Plug-in *ZCharMap* stellt wenig mehr dar als eine `JTable`, die auf Klicks des Benutzers vorher definierte Texte in ein Textfeld eingibt. Dazu kommen noch die Buttons zur Auswahl einer auszuführenden Aktion (Typecheck, Konvertieren). Diese Funktionen sind durch Aufrufe der Funktionen der entsprechenden Projekte realisiert.

5.5 Benutzung des Quellcodes

Die Benutzung des Quellcodes kann in folgende Unterbereiche und die dazugehörigen wichtigsten Klassen unterteilt werden:

allgemein: `Term`, `SectionManager`, `Markup`

Typecheck: `TypeCheckUtils`

ZML lesen/schreiben: `JaxbXmlReader` und `JaxbXmlWriter`

LaTeX und Unicode lesen/schreiben: `Source` und `PrintUtils`

Im Folgenden wird kurz auf diese Bereiche und die Klassen eingegangen.

- `Term`
  Ein Term ist ein Z-Ausdruck, der als *Annotated Syntax Tree* repräsentiert ist. Er wird entweder mit Hilfe des Parsers aus Unicode oder LaTeX gewonnen oder direkt als ZML eingelesen.

- `SectionManager`
  Der SectionManager dient als Speicherort für die Verwaltung beim Bearbeiten von Ausdrücken. So *cached* er zum Beispiel beim Parsen häufig genutzte Objekte und erhöht so die Performanz. Er hat noch weitere Funktionen, die von den Parsern und Konvertierungen genutzt werden.
  Ein neuer `SectionManager` mit leerem *Cache* wird mit Hilfe des Standardkonstruktors erzeugt. Um den Cache zu erhalten, gibt es die Methoden `clone` sowie `reset`. `clone` kopiert den gesamten zum Zeitpunkt des Aufrufs vorhandenen *Cache*, `reset` leert den Cache größtenteils und behält nur die Teile des *Toolkits* im Speicher.

- `Markup`
  Gibt an, in welchem Format ein Z-Ausdruck vorliegt. Hierzu werden die statischen Attribute `Markup.UNICODE`, `Markup.LATEX` und `Markup.ZML` benutzt.

5.5.1 Typecheck

Mit *Typecheck* ist das Überprüfen von Z-Ausdrücken möglich. Eine Object-Z-Version wird momentan entwickelt. Ein *Typecheck* ist mit Hilfe der Klasse `TypeCheckUtils` einfach zu realisieren:

```
List TypeCheckUtils.typecheck(Term t, SectionInfo si, Markup markup)
```

`Term` ist der zu überprüfende Term. Als `SectionInfo` kann ein `SectionManager` eingesetzt werden, wie zuvor beschrieben. Das `Markup` gibt an, in welcher Form man die Fehlerliste erhalten möchte. Als Rückgabewert erhält man eine Liste mit `ErrorAnn`-Objekten, die die Zeile, die Spalte sowie den Text der aufgetretenen Fehler beinhalten.

5.5.2 Lesen, Schreiben und Konvertieren von Z Darstellungen

ZML

Das Lesen eines Terms, der im ZML-Format gespeichert ist, ist mit folgendem Aufruf möglich:

```
Term JaxbXmlReader.read(InputStream is)
```

Der `Term` wird aus dem `InputStream` gelesen. Das Schreiben eines Terms als ZML in einen `Writer` ist ebenso einfach:

```
JaxbXmlWriter.write(Term t, Writer w)
```

Unicode und LaTeX

Unicode- und LaTeX-Ausdrücke werden mit Hilfe des Parsers eingelesen. Zunächst erstellt man ein `Source`-Objekt, das als Quelle für den einzulesenden Ausdruck dient. Hierzu kann entweder `FileSource`, `UrlSource` oder `StringSource` verwendet werden, welche aus einer Datei, von einer URL beziehungsweise einem String lesen. Alternativ kann von der abstrakten Klasse `Source` abgeleitet werden. Dadurch erhält man die Möglichkeit, einen beliebigen `Stream` als Quelle anzugeben. Anschließend muss noch eingestellt werden, um was für eine Art von Text es sich handelt (Unicode oder LaTeX):

```
Source.setMarkup(Markup m)
```

Als `Markup` muss `Markup.LATEX` oder `Markup.UNICODE` verwendet werden. Jetzt kann der Term bereits eingelesen werden.

```
Term t = ParseUtils.parse(Source source, SectionInfo si)
```

`SectionInfo` ist dabei wie oben beschrieben zu verwenden. Sollten sich Fehler im Text befinden, wird eine `ParseException` geworfen, welche eine Liste von `ErrorAnn`-Objekten enthält, wie bereits im Abschnitt *Typecheck* (Kapitel 5.5.1, S. 46) beschrieben.

```
PrintUtils.printUnicode(Term t, Writer w, SectionInfo si)
```

Hiermit wird das Schreiben eines Terms in Unicode in einen `Writer` realisiert. `SectionInfo` ist wie in der Beschreibung von `SectionManager` dargestellt zu verwenden. Soll die Ausgabe in LaTeX erfolgen, muss entsprechend

```
PrintUtils.printLatex(Term t, Writer w, SectionInfo si)
```

aufgerufen werden.

5.6 Lizenzen

Der gesamte Quellcode des *Community Z*-Projekts steht unter der GPL.
Die Schrift *CZTSans* steht unter dem „Bitstream Vera Fonts Copyright", das aussagt, dass die Schrift frei verwendet und verändert werden darf, solange beim Verändern nicht der Name Vera oder Bitstream verwendet wird. Die Verwendung ist erlaubt, solange nicht die Schrift als allein stehendes Produkt verkauft wird. Es handelt sich also um eine sehr offene Lizenz.
Einige externe Abhängigkeiten stehen zwar nicht unter Open-Source-Lizenzen, können aber dennoch problemlos verwendet werden.

5.7 Fazit

Das *Community Z Tools*-Projekt bietet einige durchaus brauchbare Werkzeuge. Dazu gehören die Corejava-Klassen ebenso wie der Parser und die ZML-Darstellung. Auch der Typechecker scheint einfach verwendbar zu sein. Es ist ratsam, diese bereits existierende Funktionalität soweit möglich zu nutzen. Die Benutzung der Schriftart *CZTSans* bietet sich auf Grund der freizügigen Lizenz auch an.
Die Verwendung von *ZCharMap* scheint weniger sinnvoll zu sein, da dieses Plug-in sehr einfach aufgebaut ist, aber dennoch wegen der JEdit-Anbindung vergleichsweise viel unnötigen Quellcode enthält. Eine eigene Implementierung zur Eingabe von Z-Ausdrücken wird vermutlich zu bessereren Resultaten führen.
Die Nutzung von gewissen Werkzeugen aus den *Community Z Tools* scheint also durchaus sinnvoll.

Kapitel 6

Einführung in UML und XMI

Florian Marwede

6.1 UML - Unified Modeling Language

6.1.1 Einleitung

UML ist eine standardisierte Beschreibungssprache für Strukturen und Abläufe in objektorientierten Softwaresystemen und wurde ab Ende 1997 von der *Object Management Group* (OMG) entwickelt. Sie ist eine Sammlung grafischer Notationsweisen, die in zwei Bereiche aufgeteilt ist:

1. Strukturdiagramme, um *statische* Aspekte darzustellen
2. Verhaltensdiagramme, um *dynamische* Aspekte darzustellen

Die einzelnen Diagrammtypen, die sich den beiden Bereichen zuordnen lassen, werden im nächsten Kapitel dargestellt.

UML wird durch ein Metamodel, die *Meta Object Facility* (MOF) definiert; diese wird im Wesentlichen durch vier Abstraktionsebenen beschrieben:

- M0-Ebene: konkrete, ausgeprägte Daten
- M1-Ebene: z.B. physikalische oder logische Daten-, Prozess- oder UML- bzw. Objekt-Modelle, die die Daten der M0-Ebene definieren
- M2-Ebene: Meta-Modelle, die definieren, wie die Modelle aufgebaut und strukturiert sind
- M3-Ebene: Meta-Meta-Modelle (bzw. MOF-Ebene); abstrakte Ebene, die zur Definition der M2-Ebene herangezogen wird

6.1.2 Übersicht über die einzelnen Diagrammtypen

Strukturdiagramme

- Klassendiagramm (engl. *class diagram*): Ein Klassendiagramm beschreibt sogenannte Klassen bzw. Objekt-Klassen, also programmierte „Vorbilder“ für zur Laufzeit aktive Software-Einheiten, die Objekte.
  Eine Klasse wird als Rechteck dargestellt und sie wird beschrieben durch Angaben über ihren Zustand und mögliche auf ihr durchführbaren Operationen. Den Zusammenhang zwischen mehreren Klassen kann man durch die Angabe von Relationen zwischen ihnen darstellen, welche durch verschiedene Pfeile und Linien ausgedrückt werden.

- Objektdiagramm (engl. *object diagram*): Objekte sind konkrete Instanzen und Objektdiagramme stellen somit Objekte und Relationen zwischen Objekten dar.

- Komponentendiagramm (engl. *component diagram*): Komponentendiagramme stellen Software-Komponenten und deren Schnittstellen dar. Software-Komponenten stellen in sich geschlossene Module auf einer höheren Abstraktionsebende als Klassen und Objekte dar. Eine wesentliche Anforderung an Komponenten ist eine möglichst geringere Kopplung nach außen zu anderen Bausteinen des Systems und eine möglichst hohe Kohäsion in ihrem Inneren.

- Kompositionsstrukturdiagramm (engl. *composite structure diagram*): Mit einem Kompositiobsstrukturdiagramm lassen sich die inneren Zusammenhänge innerhalb bestimmter Elemente der UML darstellen. Wesentliche innere Bestandteile in einem Kompositionsdiagramm sind die sogenannten *Parts*, die definierte Rollen übernehmen. Welcher konkrete Software-Baustein diese Rolle dann tatsächlich übernimmt, wird hier nicht betrachtet.

- Verteilungsdiagramm (engl. *deployment diagram*): In einem Verteilungsdiagramm werden typischerweise die Zusammenhänge zwischen Hard- und Software eines Systems dargestellt, also z.B. welche Teile der Software auf welchen Servern untergebracht werden.

- Paketdiagramm (engl. *package diagram*): Wenn in der Software Klassen zu Paketen zusammengefasst sind, lassen sich diese und die Zusammenhänge zwischen ihnen in einem Paketdiagramm darstellen. Obwohl dies der naheliegenste Anwendungsfall für diesen Diagrammtyp ist, lässt er sich viel allgemeiner einsetzen – z.B. können auch Aktivitäten (siehe Aktivitätsdiagramm) in Paketen zusammengefasst werden und entsprechend auch mit dem Paketdiagramm dargestellt werden.

Verhaltensdiagramme

- Anwendungsfalldiagramm (engl. *use case diagram*): Es zeigt eine bestimmte Sicht auf das erwartete Verhalten eines Systems und wird deshalb für die Spezifikation der Anforderungen an ein System eingesetzt. In einem Anwendungsfalldiagramm

werden typischerweise Anwendungsfälle und Akteure mit ihren Abhängigkeiten und Beziehungen dargestellt.

- Zustandsdiagramm (engl. *statechart*): Das UML-Zustandsdiagramm ist eines von vielen in der Informatik gebräuchlichen Notationsformen, um die Folge von Zuständen zu definieren, die ein Objekt einnehmen kann. Es werden hierfür alle Zustände, Zustandsübergänge und Bedingungen für und Konsequenzen von Zustandsübergängen in Form eines endlichen Automaten dargestellt.

- Aktivitätsdiagramm (engl. *activity diagram*): Mit einem Aktivitätsdiagramm wird häufig der Ablauf eines Anwendungsfalls beschrieben, eignet sich aber zur Modellierung aller Aktivitäten innerhalb eines Systems.

- Sequenzdiagramm (engl. *sequence diagram*): Ein Sequenzdiagramm ist eine grafische Darstellung einer Interaktion und spezifiziert den Austausch von Nachrichten zwischen Ausprägungen, die im Diagramm als Lebenslinien dargestellt sind. Zum Beispiel ist der Aufruf einer Java-Methode in diesem Sinne eine Sequenz, da nacheinander weitere Methoden (Nachrichten) von verschiedenen Objekten (Ausprägungen) aufgerufen werden.

- Interaktionsübersichtsdiagramm (engl. *interaction overview diagram*): Ist eine Interaktion zu komplex, um beispielsweise in einem Sequenzdiagramm dargestellt zu werden (da diese bei zuvielen Lebenslinien und Nachrichten zu unübersichtlich werden), können die wichtigsten Aktivitäten und Abläufe dieser Interaktion abstrakt in einem Interaktionsübersichtsdiagramm dargestellt werden. Teilinteraktionen, die sich aus diesem Diagramm ablesen lassen können, sind dann wieder als Sequenzdiagramm darstellbar.

- Kommunikationsdiagramm (engl. *communication diagram*): Kommunikationsdiagramme stellen ebenso wie Sequenzdiagramme Interaktionen mit ihren versendeten Nachrichten dar, doch liegt hier nicht der Schwerpunkt auf der Reihenfolge der versendeten Nachrichten, sondern auf den Relationen der Ausprägungen, die diese Nachrichten versenden, zueinander – eine Zeitachse ist hier also nicht gegeben, dafür können die Symbole für die Ausprägungen so angeordnet werden, dass sie für die Darstellung am übersichtlichsten sind.

- Zeitverlaufsdiagramm (engl. *timing diagram*): Zeitverlaufsdiagramme sind zweidimensionale Diagramme, bei denen die X-Achse die Zeit, und die Y-Achse Objekte (im weitesten Sinne) und deren Zustände darstellen. Dieser Diagrammtyp ähnelt damit der Anzeige eines Oszilloskops, wird also seit langem in der Elektrotechnik verwendet.

6.1.3 Drei Diagrammtypen am Beispiel erläutert

Kompositionsstrukturdiagramm

Abbildung 6.1 zeigt ein Kompositionsstrukturdiagramm für einen Software-Baustein „Wetterstation“. Diese hat drei *Parts*, also innere Bausteine: einen *Logger*, einen *Report-Generator* und einen *Data-Collector*. Der Baustein besitzt außerdem einen komplexen

Port, der die beiden Schnittstellen „Anfrage“ und „Report“ bereitstellt. Werden diese Schnittstellen verwendet, wird dies nach innen delegiert, d.h. die „Anfrage“ wird an den Logger und der „Report“ an den Report-Generator delegiert, die jeweils eine entsprechende Schnittstelle bereitstellen.
Nach demselben Prinzip hat der Baustein Wetterstation einen Port, der die Schnittstelle Temperatur benötigt. Hier können also Temperatursensoren „andocken“, um Temperaturdaten zu übermitteln. Diese werden wieder delegiert und zwar zum Part „Data-Collector.“
Intern stellen der „Logger“ und der „Report-Generator“ Schnittstellen für den „Data-Collector“ bereit, damit dieser ihnen Daten übermitteln kann.

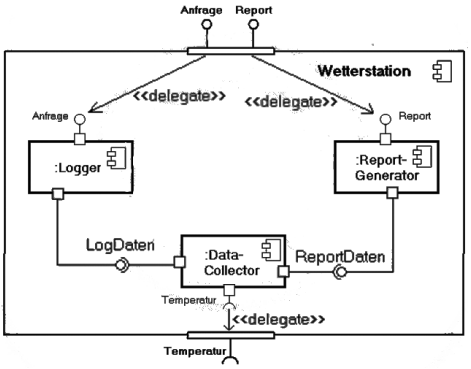


Abbildung 6.1: Ein Kompositionsstrukturdiagramm am Beispiel einer Wetterstation

Klassendiagramm

Ein einfaches Beispiel für ein Klassendiagramm ist das *Interface* „Tür“ welches die Schnittstelle für eine Türsteuerung beschreiben kann:

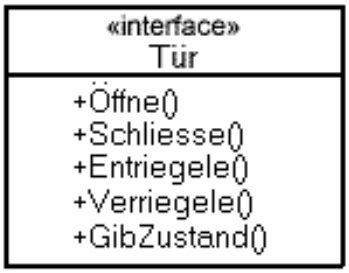


Abbildung 6.2: Ein Klassendiagramm am Beispiel einer Türsteuerung

Ein *Interface* in der UML ist ein vordefinierte Stereotyp für das Klassendiagramm, d.h. es wird die Definition des Klassendiagramms modifiziert und eingeschränkt. In diesem Fall ist die wesentliche Beschränkung das Verbieten von Attributen.

Zustandsdiagramm

Es gibt zwei Arten von Zustandsdiagrammen:

1. Protokollzustandsdiagramm
2. Verhaltenszustandsdiagramm

Das Verhaltenszustandsdiagramm stellt die Abfolge innerer Zustände eines Objektes dar, das Protokollzustandsdiagramm nur die einer Schnittstelle, diesem steht deshalb auch nur eine Untermenge der Notationsmöglichkeiten des Verhaltenszustandsdiagramm zu Verfügung.
Abbildung 6.3 zeigt ein Protokollzustandsdiagramm für die Schnittstelle einer Türsteuerung.

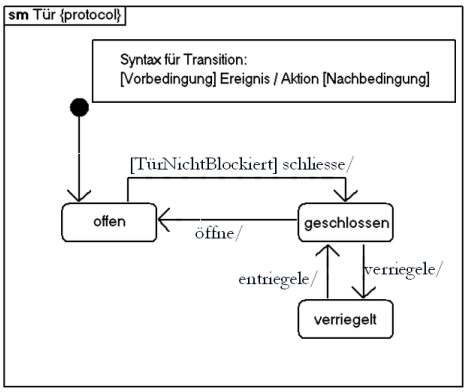


Abbildung 6.3: Protokollzustandsdiagramm am Beispiel einer Türsteuerung

Abbildung 6.4 zeigt ein Verhaltenszustandsdiagramm für eine Ampelsteuerung. Im Wesentlichen ist hier auch einer der auffälligsten Unterschiede zu den Protokollzustandsdiagrammen dargestellt: Die Möglichkeit, eine Art „Unterautomaten“ darzustellen: Beim Ereignis *start* geht das Objekt vom Zustand *aus* in den Zustand *bereit* bzw. *bereit.rot*, weil *rot* der Initialzustand von *bereit* ist. Ist das Objekt im Zustand „bereit“ – gleich in welchem Unterzustand –, und wird das Ereignis *stop* ausgelöst, geht das Objekt wieder in den Zustand *aus*.

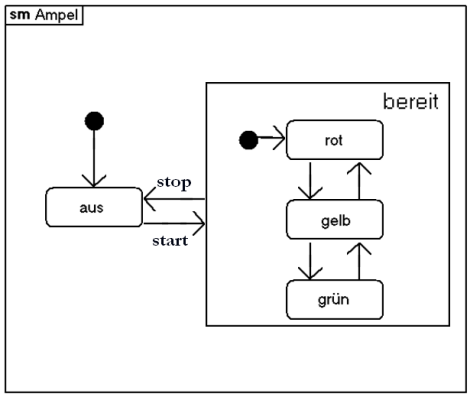


Abbildung 6.4: Verhaltenszustandsdiagramm am Beispiel einer Ampelsteuerung

Die Beispiele für die Erläuterung der UML-Diagramme wurden entnommen aus [PBG04].

6.1.4 Wesentliche Neuerungen in der UML 2.0

- Konsolidierung des Metamodels: MOF und UML waren vorher unabhängig voneinander. Dadurch, dass es nun ein durchgängiges Metamodell für die UML gibt, soll es möglich sein, individuelle Anpassungen an der UML vorzunehmen, um z.B. Codegenerierung für bestimmte Anwendungsgebiete zu erleichtern.

- Neue Konzepte (*Parts*, *Ports*, Konnektoren, Interaktion): Aufgrund der Erfahrungen, die mit vorherigren UML-Versionen gemacht wurden, sind einige neue Aspekte hinzugefügt worden, um das Modellieren anwendungsnäher gestalten zu können.

- Neue Notationen (*CompositePart*, Interaktionsübersicht, Zeitverlauf): Demselben Gedanken geschuldet sind neue Diagrammtypen, die gerade im Darstellen dynamischer Abläufe das Sequenzdiagramm entlasten sollen, welches nicht für alle Anwendungen optimal ausgelegt ist (vgl. z.B. Einführung des Interaktionsdiagramms, Abschnitt 6.1.2, S. 51).

6.2 XMI - XML Metadata Interchange

6.2.1 Einleitung

XMI gehört zur Sprachfamilie der *eXtensible Markup Language* (XML), d.h. ein XMI-Dokument ist immer auch ein XML-Dokument, und ein XMI-Schema ist immer auch ein XML-Schema.
In einem XML-Dokument werden Daten gespeichert, in einem XML-Schema wird festgelegt, wie diese Daten auszusehen haben. Man kann also XML-Dokumente gegen XML-Schema „laufen lassen“, um zu prüfen, dass ihre Daten die korrekte Form, das korrekte Format haben.
XMI ist eine Spezialisierung hiervon. Bei XMI geht es nicht allgemein um Daten, sondern um Objekte bzw. um das, was in der MOF unter dem Begriff verstanden wird, denn dies ist das Metamodell für XMI – womit die Brücke zwischen UML und XMI geschlagen ist.
In einem XMI-Dokument werden also Objekte gespeichert, während in einem XMI-Schema festgelegt wird, wie diese Objekte auszusehen haben. Dies kann unter anderem beliebig konkret ausgedrückt werden. Zum Beispiel kann festgelegt werden, dass alle Objekte keine Attribute haben dürfen, oder dass sie generell welche haben dürfen. Aber auch, dass sie z.B. nur eine bestimmte Anzahl eines bestimmten Typs haben dürfen usw.

6.2.2 Ein Überblick über XML

Die allgemeine Form eines XML-Elements ist:

```
<Name Attribute>Inhalt</Name>
```

Ein XML-Element besteht mindestens aus einem sogenannten *Tag*: `<Name/>`. Hat dieser *Tag* einen Inhalt, wird er in zwei *Tags* „aufgespalten“, einen öffnenden und einen schließenden *Tag*: `<Name>Hallo Welt</Name>`. Attribute sind eine optionale Liste von Name-Wert-Paaren: `<Name farbe="rot">Hallo Welt</Name>`. XML-Elemente können baumartig ineinander verschachtelt werden, wobei der schließende *Tag* eines inneren Elements nie nach dem schließenden *Tag* des dieses Element umschließenden *Tags* kommen darf. Außerdem ist nicht viel nötig, um XML-Dokumente verstehen zu können: Es gibt Namensräume, in denen Namen eindeutig sein müssen, Regeln, die definieren, was für Zeichen in Namen vorkommen dürfen und eine Möglichkeit, *Tag* als *plain text* zu schreiben, also uninterpretiert zu lassen.
XML-Schemata jedoch haben viele weitere Syntaxelemente – diese zu erklären, ist nicht Ziel dieses Abschnitts. Es sei nur soviel gesagt, dass XML-Schemata auch XML-Dokumente in dem Sinne sind, dass sie der grundlegenden XML-Syntax (die eben angedeutet beschrieben wurde) folgt. Jedoch gibt es hier viele definierte Element- und Attributnamen mit einer festgelegten Semantik.
Schließlich sei noch erwähnt, dass die sogenannten *Document Type Definitions* (DTDs) eine veraltete Möglichkeit sind, die Struktur und das Format von XML-Dokumenten festzulegen. Sie haben eine eigene Syntax, die *nicht* auf XML aufbaut.

6.2.3 Ein Beispiel

Um mit XMI sinnvoll arbeiten zu können, sind Grundkenntnisse der XML-Schemadefinitionen notwendig. Diese können zum Beispiel unter [SMT06] erworben und vertieft werden.
Nehmen wir an, es soll eine Anwendung für eine Autovermietung modelliert werden und in einem dabei geschaffenen Klassendiagramm gibt es eine Klasse *Car*. Diese soll einfach nur ein Attribut `available` vom Typ `boolean` haben. Würde dieses Klassendiagramm nach XMI importiert, könnte der entsprechende Code so aussehen:

```
<xsd:complexType name="Car">
  <xsd:element name="available" type="xsd:boolean"/>
  <xsd:element minOccurs="0" maxOccurs="unbounded" ref="xmi:Extension"/>
</xsd:complexType>
```

Eine Klasse wird immer als `complexType` definiert. In einem `complexType`-Element, ist es möglich, durch ein `element`-Element ein Attribut der Klasse anzugeben. Mit `minOccurs` und `maxOccurs` kann man Unter- und Obergrenzen festlegen, wieviele solcher Elemente auftauchen dürfen. Die *default*-Werte für beide sind „1“, können also im Fall von unserem Attribut *available* weggelassen werden.
Wir sehen aber auch, dass dieser `complexType` null oder beliebig viele sogenannte `extensions` haben kann. Dies dient dem Zweck, spezifische, werkzeugabhängige Informationen zu speichern, also z.B. Angaben, an welcher Stelle die entsprechende Klasse im Diagramm eingezeichnet war.
Folgendes XML-Dokument ist also korrekt im Sinne dieses Schemas:

```
<Car>
```

```
    <available>true</available>
  </Car>
```

Zur Vertiefung des Themas XMI wird [GDB02] empfohlen.

Kapitel 7

CSP-OZ: Kombination zweier Methoden

Janna Arnold

7.1 Einführung

CSP-OZ ist eine Kombination aus den beiden Spezifikationssprachen CSP [Hoa04] und Object-Z [Smi00]. Um CSP-OZ verstehen zu können, werden zunächst die einzelnen Spezifikationssprachen vorgestellt. CSP dient zur Beschreibung reaktiver Systeme. Das sind Systeme, die unabhängig voneinander arbeiten und gelegentlich miteinander kommunizieren. Z spezifiziert sequentielle Systeme, die durch Zustandsräume und deren Veränderung gekennzeichnet sind.
CSP-OZ kombiniert die Vorteile der beiden Sprachen, nach [Fis00]. Dadurch können reaktive Systeme mit einer Prozess- und einer Datensicht spezifiziert werden.

7.2 Die Spezifikationssprache CSP

7.2.1 Einleitung

CSP wurde von C.A.R Hoare, Universität Oxford, eingeführt. CSP steht für *Communicating Sequential Processes*. Wie in [Hoa04] und [Old05] zu lesen, dient diese Spezifikationssprache zur Beschreibung von Systemen, die aus mehreren Komponenten bestehen, die parallel arbeiten und gelegentlich miteinander kommunizieren. Abbildung 7.1 zeigt ein reaktives System. Beispielsweise sind ein Fahrstuhl und ein Nutzer eines Fahrstuhls (Fahrgast) einzelne sequentielle Prozesse, die parallel arbeiteten und kommunizieren, wenn der Fahrgast einen Knopf drückt und so ein Signal über den Kanal *knopfDruecken* an den Prozess *Fahrstuhl* sendet.

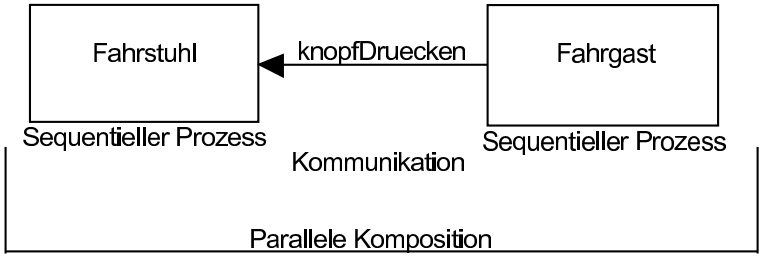


Abbildung 7.1: Beispiel für ein reaktives System

7.2.2 Syntax

Um die Sprache zu verstehen, muss zunächst die Syntax festgelegt werden. Dazu werden hier einige Symbole und Mengen vorgestellt:

$a, b \in \mathit{Events}$ (Ereignisse)
$c.v \in \mathit{Events}$ ist eine Kommunikation, wobei der Wert v über den Kanal c gesendet wird
$A, B \subseteq \mathit{Events}$ (Alphabete)
$\tau \notin \mathit{Events}$ (Symbol für die interne Aktion)
$\surd \notin \mathit{Events}$ (Symbol für die Terminierung)
$\alpha, \beta \in \mathit{Act} = \mathit{Events} \cup \{\tau, \surd\}$ (Alle Aktionen)
$R : \mathit{Events} \longleftrightarrow \mathit{Events}$ (Relation zur Umbenennung von Ereignissen)
$X, Y \in \mathit{Idf}$ (Prozessidentifikatoren)
$P, Q \in \mathit{Proc}$ (CSP-Prozess)

Mit Hilfe dieser Symbole lässt sich nun die Syntax eines CSP-Prozesses in Backus-Naur-Form aufschreiben:

$$P ::= \mathit{STOP} \mid \mathit{SKIP} \mid a \rightarrow Q \mid P;\ Q \mid P \underset{\{A\}}{\parallel} Q \mid P_{\{A\}}\|_{\{B\}} Q$$
$$\mid P \,|||\, Q \mid P \sqcap Q \mid P \,\Box\, Q \mid P \backslash A \mid P[R] \mid X$$

Die Bedeutung dieser Syntax wird im folgenden Kapitel mit Hilfe der informellen Semantik erklärt.

7.2.3 Informelle Semantik

Die informelle Semantik gibt der CSP-Syntax eine Bedeutung mit Hilfe von Zustandsgraphen. Einige Graphen sollen hier als Beispiel dienen, um die grundlegende Bedeutung der Syntax zu geben.

Abbildung 7.2 zeigt den CSP-Prozess *STOP*.

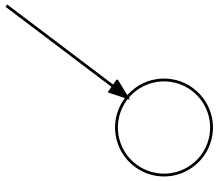


Abbildung 7.2: Der CSP-Prozess *STOP*

Abbildung 7.3 zeigt den CSP-Prozess *SKIP*.

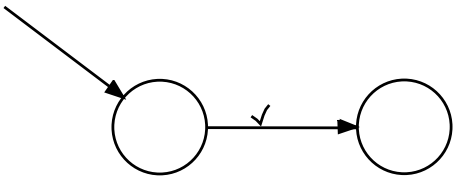


Abbildung 7.3: Der CSP-Prozess *SKIP*

Abbildung 7.4 zeigt den CSP-Prozess:

$$knopf1 \rightarrow fahreZu1 \rightarrow STOP.$$

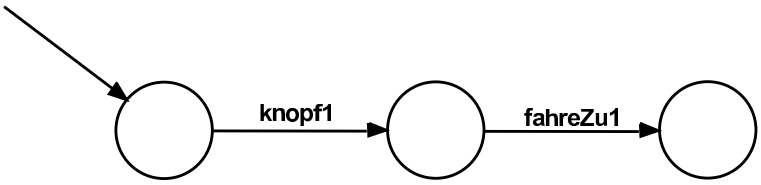


Abbildung 7.4: Der CSP-Prozess $knopf1 \rightarrow fahreZu1 \rightarrow STOP$

Abbildung 7.5 zeigt den CSP-Prozess:

$$\begin{aligned}&(knopf1 \rightarrow fahreZu1 \rightarrow STOP)\\ \Box\ &(knopf2 \rightarrow fahreZu2 \rightarrow STOP).\end{aligned}$$

Abbildung 7.6 zeigt den CSP-Prozess:

$$\begin{aligned}FS = &(knopf1 \rightarrow fahreZu1 \rightarrow FS)\\ \Box\ &(knopf2 \rightarrow fahreZu2 \rightarrow FS).\end{aligned}$$

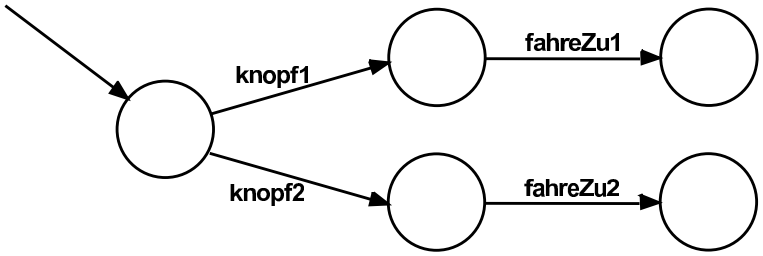


Abbildung 7.5: CSP-Prozess mit □

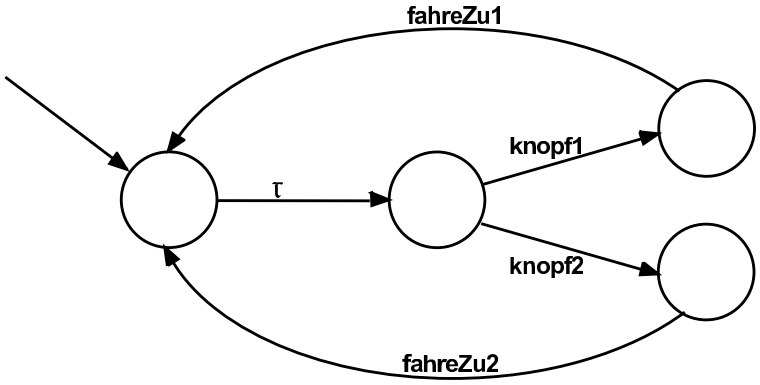


Abbildung 7.6: Der CSP-Prozess *FS*

Um die parallele Komposition zu erklären, wird ein zweiter Prozess benötigt. Der Prozess *FG* soll einen Fahrgast verdeutlichen, der den Fahrstuhl nutzt:

$$FG = einsteigen \rightarrow knopf2 \rightarrow aussteigen \rightarrow FG$$

Abbildung 7.7 stellt die parallele Komposition dar. Zunächst werden die einzelnen Prozesse dargestellt, dann die Komposition der beiden. Die Prozesse können unabhängig voneinander laufen, außer eine Aktion aus dem Synchronisationsalphabet tritt ein. Diesen Schritt müssen die Prozesse gemeinsam gehen.

$$P_4 = FG \underset{\{knopf1, knopf2\}}{\parallel} FS$$

Verwendet man die nichtdeterministische Auswahl ⊓ entsteht ein fehlerhafter Fahrstuhl *FFS*. Seine Fehlfunktion zeigt die paralle Komposition in Abbildung 7.8. Der Prozess *FFS* kann in einen Zustand wechseln, von dem aus er nicht mehr auf das Signal *knopf2* eingehen kann.

$$\begin{aligned} FFS &= (knopf1 \rightarrow fahreZu1 \rightarrow FFS) \\ &\quad \sqcap (knopf2 \rightarrow fahreZu2 \rightarrow FFS) \\ P_5 &= FG \underset{\{knopf1, knopf2\}}{\parallel} FFS \end{aligned}$$

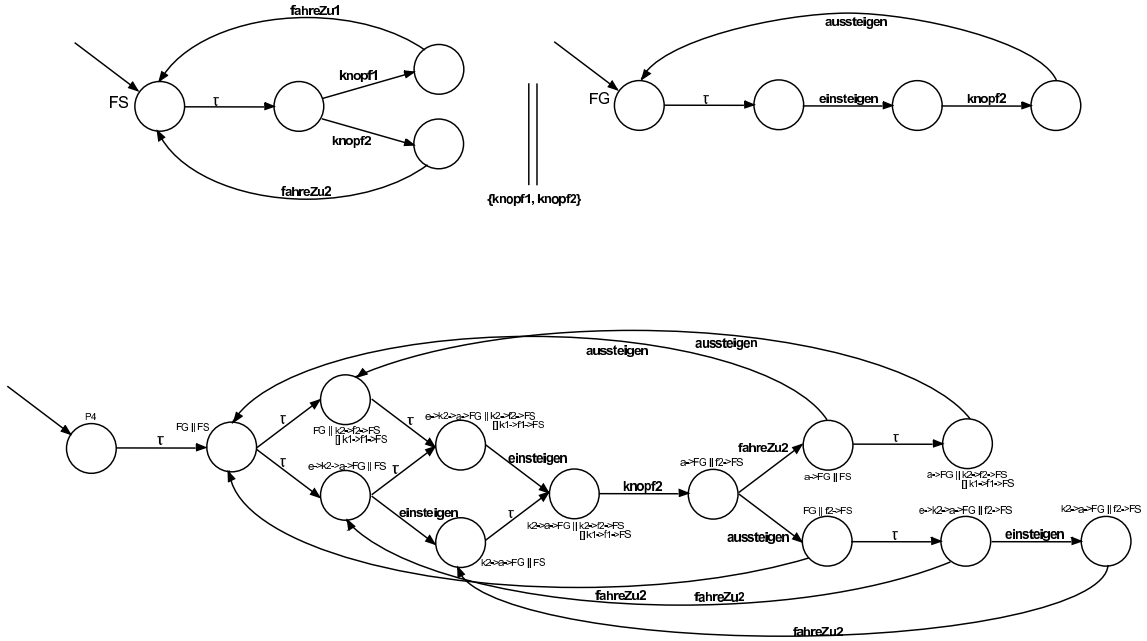


Abbildung 7.7: Der CSP-Prozess P_4

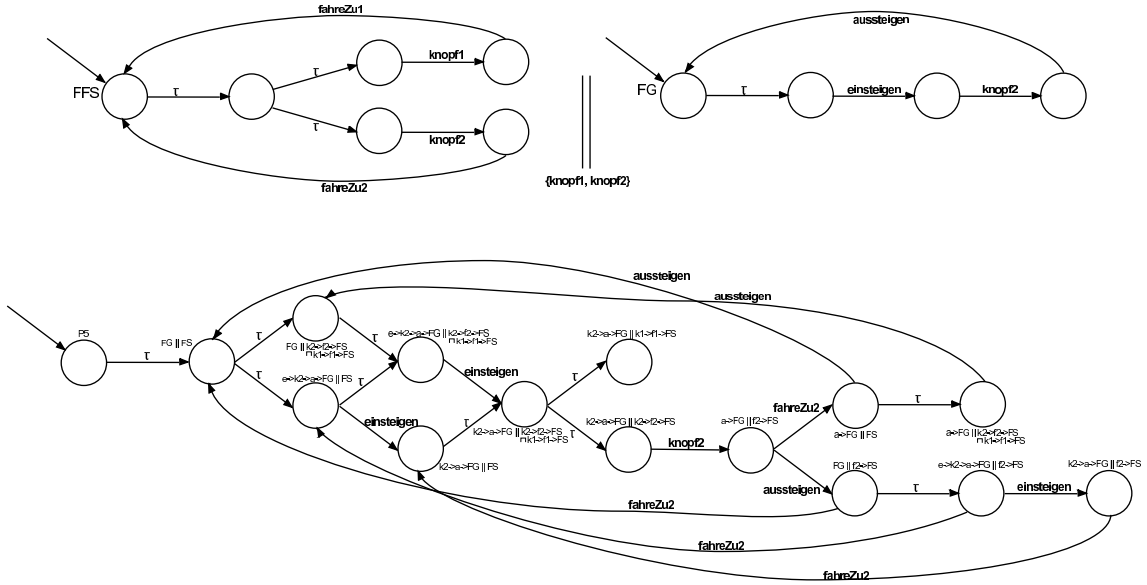


Abbildung 7.8: Der CSP-Prozess P_5

7.2.4 CSP-Semantiken

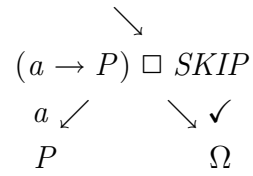
Um der Syntax von CSP Bedeutungen zuzuordnen, können verschiedene Semantiken verwendet werden. Einige sollen im Folgenden vorgestellt werden.

Strukturierte operationelle Semantik (SOS)

Für jeden CSP-Prozess werden die Zustände und Transitionen eines Automaten beschrieben. Es handelt sich um nichtdeterministische Automaten mit τ-Transitionen der Form $\mathbf{A} = (A, S, \rightarrow, s_0)$, mit:
$A \in \mathit{Events}$
S eine möglicherweise unendliche Folge von Zuständen, die CSP-Prozesse (Proc) sind
$\rightarrow \subseteq \mathit{Proc} \times \mathit{Act} \times \mathit{Proc}$
$s_0 \in S$ ist der Startzustand

Die Transitionen lassen sich nach vorgegebenen Regeln herleiten, welche zum Beispiel in [Old05], [Ros97] nachzulesen sind.

Beispiel:



Dabei drückt Ω die gerade eingetretene Terminierung aus.

Trace-Semantik

Die *Trace*-Semantik gibt die einfache Sicht auf CSP-Prozesse. Eine *Trace* ist eine endliche Sequenz sichtbarer *Events* ($\mathit{Act}\backslash\{\tau\}$). Die *Trace*-Semantik eines Prozesses ist die Menge aller mögliche *Traces*.
Die Beispiele in Abbildung 7.9 und 7.10 zeigen einen CSP-Prozess in *Trace*-Semantik und strukturierter operationeller Semantik.

$$T((a \rightarrow \mathit{SKIP} \,\Box\, c \rightarrow b \rightarrow \mathit{SKIP})\backslash c) = \{<>, < a >, < a, \checkmark >, < b >, < b, \checkmark >\}$$

Abbildung 7.9: CSP-Prozess in Trace-Semantik

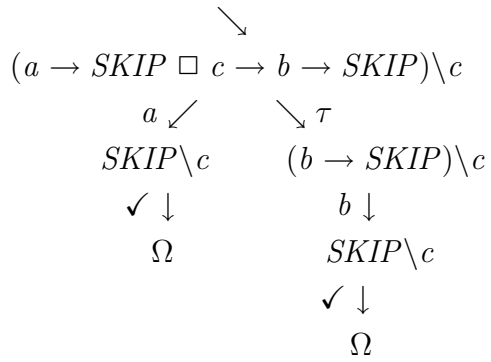


Abbildung 7.10: CSP-Prozess in SOS

Failure-Divergences-Semantik

Die *Failure-Divergences*-Semantik (FD-Semantik) gibt eine genauere Sicht auf die CSP-Prozesse. Unterschiedliche Prozesse können *Trace*-äquivalent sein und wären so in der *Trace*-Semantik nicht zu unterscheiden. Der Unterschied zwischen der deterministischen und der nichtdeterministischen Auswahl ist in der *Trace*-Semantik nicht erkennbar.
Die FD-Semantik gibt zu jedem Prozess ein Paar aus der Menge aller Fehler und der Menge aller Divergenzpunkte an ($FD(P) = (F(P), D(P))$).
Ein Fehler ist ein Paar aus einer *Trace* und der Menge der abgelehnten Aktionen.
Beispiel:

$$\begin{aligned} F((a \to STOP \;\Box\; c \to b \to STOP)\backslash c)) = \; & \{(<>, \varnothing), (<>, \{a\}), (< a >, \varnothing), (< a >, \{a\}), \\ & (< a >, \{b\}), (< a >, \{a, b\}), (< b >, \varnothing), \\ & (< b >, \{a\}), (< b >, \{b\}), (< b >, \{a, b\})\} \end{aligned}$$

Ein Divergenzpunkt ist eine *Trace*, nach der eine unendliche Folge von τ-Transitionen möglich ist.

Genauer soll hier jedoch auf die CSP-Semantiken nicht eingegangen werden.

7.3 Die Spezifikationssprache Z

Die Spezifikationssprache Z spezifiziert Daten. Sequentielle Systeme werden über Zustandsräume und Zustandstransitionen beschrieben.
In Z werden mathematische Ausdrücke verwendet. Hierzu zählen

- Mengenausdrücke der Form $\{Deklaration \mid Praedikat \bullet Ausdruck\}$
- Axiomatische Ausdrücke

- Relationale Ausdrücke und Funktionen
- Aussagenlogische und prädikatenlogische Ausdrücke

7.3.1 Z-Schema

Daten werden in einem Z-Schema dargestellt, um sie zu strukturieren [JW96]. Dieses hat entweder die Form $Name \mathrel{\widehat=} [Deklaration \mid Praedikat]$ oder wird als Schema-Box dargestellt. Diese hat ebenfalls die Bestandteile *Name*, *Deklaration* und *Praedikat*. In der folgenden Abbildung 7.11 wird die Datenhaltung eines Fahrstuhls als Z-Schema-Box dargestellt.

Fahrstuhl

$$aktuellerStock : \mathbb{N}$$

$$aktuellerStock \geq 0$$
$$aktuellerStock \leq 3$$

Abbildung 7.11: Z-Schema Fahrstuhl

7.3.2 Transitionen

In Z sollen nicht nur Datenzustände beschrieben werden, sie sollen auch verändert werden können. Um die Wirkung von Operationen auf die Variablen eines Zustandsraumes auszudrücken, werden zusätzliche Versionen eines Zustandsraums eingeführt, die dekoriert werden, vgl. Abbildung 7.12.

Fahrstuhl′

$$aktuellerStock' : \mathbb{N}$$

$$aktuellerStock' \geq 0$$
$$aktuellerStock' \leq 3$$

Abbildung 7.12: Dekoriertes Z-Schema Fahrstuhl

Operationen werden ebenfalls als Z-Schema dargestellt. Wir betrachten in Abbildung 7.13 die Operation *fahreZu*, die den aktuellen Stock des Fahrstuhls verändert.
Da ungestrichene und gestrichene Variablen in allen Z-Schemata für Operationen vorkommen, darf man verkürzt den Δ-Operator verwenden.
Das Schema darf verkürzt wie in Abbildung 7.14 geschrieben werden.

fahreZu

$aktuellerStock : \mathbb{N}$
$aktuellerStock' : \mathbb{N}$
$neu? : 0..3$

$aktuellerStock \geq 0$
$aktuellerStock \leq 3$
$aktuellerStock' \geq 0$
$aktuellerStock' \leq 3$
$aktuellerStock' = neu?$

Abbildung 7.13: Z-Schema der Operation *fahreZu*

fahreZu

$\Delta Fahrstuhl$
$neu? : 0..3$

$aktuellerStock' = neu?$

Abbildung 7.14: Z-Schema der Operation *fahreZu* mit Δ-Operator

7.4 Die Spezifikationssprache CSP-OZ

CSP-OZ ist eine Kombination der Spezifikationssprachen CSP und Object-Z. Mit CSP-OZ ist eine Spezifikation reaktiver Systeme mit einer Prozess und einer Datensicht möglich. Der Vorteil der Kombination ist, dass Kommunikationen im CSP-Teil Auswirkungen auf Datenzustände im OZ-Teil haben können. Somit sind CSP-OZ-Klassen im Vergleich zu Object-Z-Klassen aktiv. Am Beispiel des Fahrstuhls soll in Abbildung 7.15 das Zusammenwirken gezeigt werden.

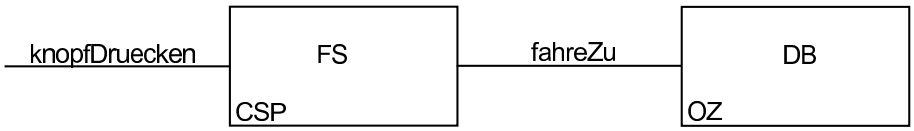


Abbildung 7.15: Z-Schema der Operation *fahreZu*

Die Kommunikation *knopfDruecken* hat keinen Einfluss auf die Datenhaltung des Fahrstuhls. Die Kommunikation *fahreZu* hingegen verändert den gespeicherten aktuellen Stock, der CSP-Teil muss mit dem Object-Z-Teil kommunizieren.

7.4.1 CSP-OZ-Klassen

In Abbildung 7.16 ist das Format einer CSP-OZ-Klasse gezeigt. Sie besteht aus einem Interface, in dem alle Kanäle deklariert werden. Darauf folgt der CSP-Teil und dann der (Object)-Z-Teil. Z besteht aus einem namenlosen Schema, das mit *State* bezeichnet wird, einem Schema `Init` für den Initialzustand und Schemata zu den Kommunikationen `com`(c).

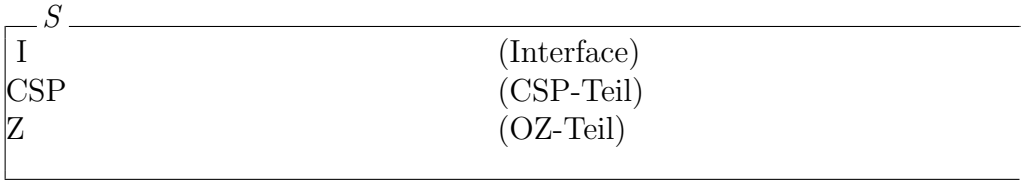


Abbildung 7.16: CSP-OZ-Schema

Das Beispiel des Fahrstuhls, das auch schon zur Erklärung von CSP-Prozessen und Z-Schemata verwendet wurde, soll nun in Abbildung 7.17 die Kombination der beiden Spezifikationssprachen verdeutlichen.

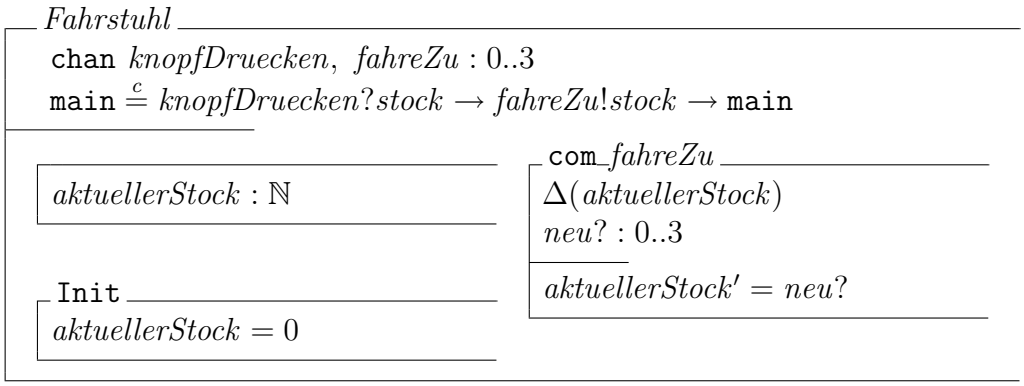


Abbildung 7.17: CSP-OZ-Schema eines Fahrstuhls

7.4.2 Semantik

Eine CSP-OZ-Spezifikation soll als Spezifikation eines Prozesses aufgefasst werden und bekommt daher eine Semantik nach dem *Failure − Divergences*-Modell der CSP-Prozesse. Dazu werden *FD*-Semantiken für den CSP- und den OZ-Teil gebildet und aus der parallelen Komposition dieser beiden entsteht die *FD*-Semantik einer kompletten CSP-OZ-Spezifikation. Wird eine Kommunikation im CSP-Teil ausgeführt, muss die entsprechende Operation auf den Zustand im OZ-Teil ausgeführt werden.

7.5 Fazit

CSP-OZ kombiniert die Vorteile der Spezifikationssprachen CSP und Object-Z. Dadurch können reaktive Systeme mit einer Prozess- und einer Datensicht spezifiziert werden.
Da CSP eine präzise formale Semantik hat, kann es bei einem endlichen Datenbereich, also einer endlichen Zustandsmenge, in den Dialekt CSP_M für maschinenlesbares CSP übersetzt werden. Dieser kann vom Modelchecker FDR für CSP [FDR] gelesen werden. Damit ist es möglich, CSP und unter Umständen auch CSP-OZ auf *Deadlocks*, *Livelocks* und Determinismus zu überprüfen.

Kapitel 8

CSP-OZ-DC und PEA

Jan-David Quesel

8.1 Einleitung

Verifikation ist eine Thematik, die in der heutigen Zeit immer mehr Gewicht bekommt. Spezifikationssprachen sind meist jedoch nur für einen Kontext entwickelt. Um die Vorzüge mehrerer Spezifikationssprachen nutzen zu können, gibt es Ansätze, diese zu kombinieren. Ein Ansatz ist die Kombination der Sprachen CSP und Object-Z, genannt *CSP-OZ*. Es wird ermöglicht, Komunikationen und Zustandsänderungen in CSP zu modellieren und zusätzlich die Vorzüge von Object-Z in der Spezifikation von Daten und Zustandsräumen zu verwenden. Bislang fehlt dabei die Möglichkeit, zeitliche Eigenschaften von Systemen zu spezifizieren. Hierfür bietet sich der Duration Calculus (DC) [ZHR91] an. Daraus ergibt sich die Spezifikationssprache *CSP-OZ-DC* [HM05]. Um komfortabel Model-Checking [CGP99] durchführen zu können, benötigt man eine einheitliche Semantik. Diese wird hier durch Phasen-Event-Automaten [HM05] geliefert.

8.2 Duration Calculus

Der Duration Calculus ist eine Logik über Zeitintervallen.

8.2.1 Syntax

Die Syntax des DC gliedert sich in drei Teile.

Zustandszusicherungen

Eine Zustandszusicherung π hat folgenden Aufbau:

$$\pi ::= 0 \mid 1 \mid X = d \mid \neg\pi \mid \pi_1 \wedge \pi_2$$

Hierbei ist X eine Observable. Observablen werden verwendet um Systemeingaben und Systemausgaben zu modellieren.

Terme

Ein Term θ hat folgenden Aufbau:

$$\theta ::= x \mid \ell \mid \int \pi \mid f(\theta_1, \cdots, \theta_n)$$

Hierbei ist x eine Variable, ℓ ein spezielles Symbol, π eine Zustandszusicherung und f ein Funktionssysmbol.

Formeln

Formeln können nach folgender Vorschrift gebildet werden:

$$F ::= p(\theta_1, \cdots, \theta_n) \mid \neg F \mid F_1 \wedge F_2 \mid \forall\, x : F \mid F_1;\ F_2$$

Hierbei ist p ein Prädikatssymbol, θ_1 bis θ_n Terme und x eine Variable.

8.2.2 Semantik

Die Semantik des DCs definieren wir über eine Interpretationsfunktion.

Zustandszusicherungen

Die Semantik von Zustandszusicherungen ist gegeben durch eine Funktion $\mathcal{I}$. Sie ordnet einer Zustandszusicherung zu jedem Zeitpunkt einen boolschen Wert (0 für falsch oder 1 für wahr) zu.

$$\begin{gathered}
\mathcal{I}[\![0]\!](t) = 0 \\
\mathcal{I}[\![1]\!](t) = 1 \\
\mathcal{I}[\![X = d]\!](t) = \begin{cases} 1, & \text{gdw } X = d \\ 0, & \text{sonst} \end{cases} \\
\mathcal{I}[\![\neg\pi]\!](t) = \begin{cases} 1, & \text{gdw } \mathcal{I}[\![\pi]\!](t) = 0 \\ 0, & \text{sonst} \end{cases} \\
\mathcal{I}[\![\pi_1 \wedge \pi_2]\!](t) = \begin{cases} 1, & \text{gdw } \mathcal{I}[\![\pi_1]\!](t) = 1 \text{ und } \mathcal{I}[\![\pi_2]\!](t) = 1 \\ 0, & \text{sonst} \end{cases}
\end{gathered}$$

Terme

Die Semantik von Termen wird über ebenfalls durch eine Funktion $\mathcal{I}$ geliefert, die jedem Term, gegeben eine Belegung der Variablen $\mathcal{V}$ und ein Zeitinterval $[b, e]$, eine reelle Zahl zuordnet.

$$\begin{gathered}
\mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) = \mathcal{V}(x) \\
\mathcal{I}[\![\ell]\!](\mathcal{V}, [b, e]) = e - b \\
\mathcal{I}[\![\int P]\!](\mathcal{V}, [b, e]) = \int_b^e P_{\mathcal{I}}(t)dt \\
\mathcal{I}[\![f(\theta_1, \cdots, \theta_n)]\!](\mathcal{V}, [b, e]) = \hat{f}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e]))
\end{gathered}$$

Formeln

Formeln wird über eine Funktion $\mathcal{I}$, gegeben eine Belegung der Variablen $\mathcal{V}$ und einem Zeitintervall $[b, e]$, einen boolschen Wert zugeordnet.

$$\mathcal{I}[\![p(\theta_1, \cdots, \theta_n)]\!](\mathcal{V}, [b, e]) = \hat{p}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e]))$$
$$\mathcal{I}[\![\neg F]\!](\mathcal{V}, [b, e]) = \text{tt gdw } \mathcal{I}[\![F]\!](\mathcal{V}, [b, e]) = \text{ff}$$
$$\mathcal{I}[\![F_1 \wedge F_2]\!](\mathcal{V}, [b, e]) = \text{tt gdw } \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, e]) = \text{tt und } \mathcal{I}[\![F_2]\!](\mathcal{V}, [b, e]) = \text{tt}$$
$$\mathcal{I}[\![\forall\, x : F]\!](\mathcal{V}, [b, e]) = \text{tt gdw für alle } d \in \mathbb{R} \mathit{gilt}:$$
$$\mathcal{I}[\![F]\!](\mathcal{V}[x := d], [b, e]) = \text{tt } \mathcal{I}[\![F_1;\ F_2]\!](\mathcal{V}, [b, e]) = \text{tt gdw es ein } m \in [b, e] \text{ gibt, so dass } \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, m]) = \text{tt und } \mathcal{I}[\![F_2]\!](\mathcal{V}, [m, e]) = \text{tt}$$

Abkürzungen

Um Formeln kürzer, und damit einfacher zu lesen, aufschreiben zu können.

$$\text{Punktintervall: } \lceil\rceil := \ell = 0$$
$$\text{Überall: } \lceil\pi\rceil := \ell > 0 \wedge \int \pi = \ell$$
$$\text{Irgendwann: } \Diamond F := \mathit{true};\ F;\ \mathit{true}$$
$$\text{Immer: } \Box F := \neg\Diamond\neg F$$

8.2.3 Beispiele

Im Folgenden werden einige Beispiele angeführt um die Semantik des Duration Calculus verständlich zu machen. Die Beispiele orientieren sich hierbei an den Operatoren, die nicht bereits aus der Prädikatenlogik erster Ordnung bekannt sind.
Die beiden neuen Operatoren sind $\int$ und **;**. Der Operator $\int$ wird verwendet, um Bereiche, in denen ein Term erfüllt ist, aufzusummieren und der Operator **;** dient zum Modellieren von nacheinander stattfindenden Ereignissen.
Die Formel $\int X = \ell$ beispielsweise ist so zu verstehen: Auf dem gesamten Beobachtungsintervall ist die Observable X wahr. Es kann nur punktuelle Ausnahmen geben, diese können vom DC nicht eingeschränkt werden, da über Variablen nur mithilfe des Integraloperators geredet werden kann.
Die Formel $\lceil P\rceil\,;\lceil Q\rceil$ besagt, dass es einen Zeitpunkt in unserem Beobachtungsintervall gibt, so dass vom Anfang unserer Beobachtung bis zu diesem Zeitpunkt (fast) überall P wahr ist und ab diesem Zeitpunkt Q (fast) überall wahr ist. Hierbei wird nicht ausgeschlossen, dass Q beispielweise dauerhaft wahr ist. Ebenso wird von P nicht verlangt, dass sich der Zustand der Observablen in dem sogenannten chop-Punkt ändert. Zur Veranschaulichung dieser Formel ist eine erfüllende Interpretation in Abbildung 8.1 zu sehen. Hierbei sind an den Stellen, wo keine Werte für die Observablen angegeben sind, diese beliebig.

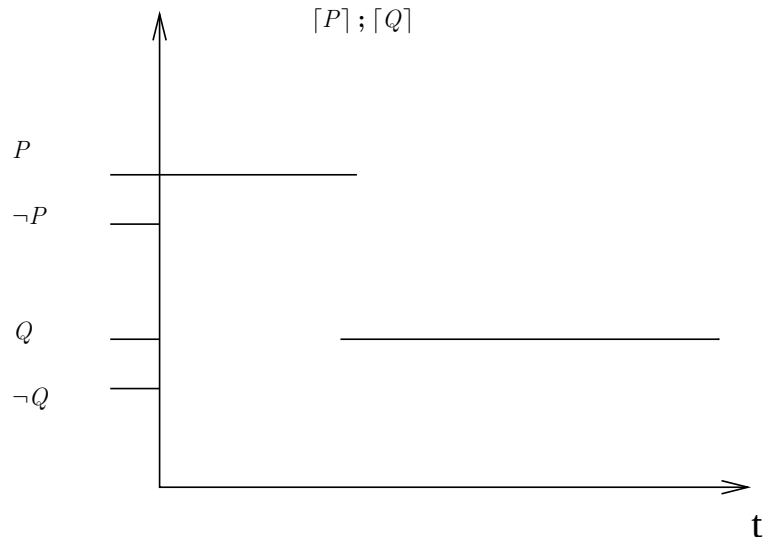


Abbildung 8.1: Erfüllende Interpretation für $\lceil P \rceil \,;\, \lceil Q \rceil$

8.3 Phase Event Automata

Phaseneventautomaten (PEA) liefern einen Formalismus, mit dem es möglich ist, eine gemeinsame Semantik für CSP-OZ-DC zu formulieren. Ein PEA ist ein 8-Tupel $(P, V, A, C, E, s, I, P_0)$. Die einzelnen Komponenten sind:

- P: Zustandsmenge
- V: Menge von Zustandsvariablen
- A: Menge von Events
- C: Menge der Uhren
- $E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathcal{P}(C) \times P$: Transitionsrelation
- $s : P \to \mathcal{L}(V)$: Invarianten-Prädikat für Zustände
- $I : P \to \mathcal{L}(C)$: Uhreninvariante für Zustände
- $P_0 \subseteq P$: Startzustände.

8.4 CSP-OZ-DC

Mit dieser Syntax-Definition der Phasen-Event-Automaten ist es uns nun möglich, eine Übersetzung der einzelnen Teile der CSP-OZ-DC-Spezifikation in PEA anzugeben.

8.4.1 CSP-Teil

Den CSP-Teil übersetzen wir unter Verwendung der operationellen Semantik. Die Zustände werden weiterhin mit CSP-Prozessen beschriftet. Als Alphabet nutzen wir das vom `main`-Prozess. Zustandsvariablen oder Uhren werden keine eingeführt. Für jede Transition aus der operationellen Semantik des `main`-Prozesses führen wir eine Transition in unserem PEA ein:

- $(p, a \wedge \bigwedge_{e \in A \setminus \{a\}} \neg e, \varnothing, p') \in E$ falls $a \neq \tau$
- $(p, \bigwedge_{e \in A} \neg e, \varnothing, p') \in E$ sonst

Zusätzlich erhält jeder Zustand eine Schleife mit: $(p, \bigwedge_{e \in A} \neg e, \varnothing, p) \in E$. Dies ist später wichtig für die parallele Komposition. Die Startzustände werden vom `main`-Prozess abgeleitet.

8.4.2 Object-Z-Teil

Zur Übersetzung des Object-Z-Teils erstellen wir einen Automaten mit nur einem Zustand. Die Startkante beschriften wir mit den Prädikaten aus `Init`. Für jedes Event wird genau eine Kante erstellt, die genau dieses Event zulässt, alle Variablen, die nicht in der Δ-Liste sind, konstant hält und die Variablen in der Δ-Liste wie im Kommunikationsschema angegeben beschränkt. Wiederum muss für die spätere parallele Komposition eine Schleife, ohne Events oder Variablenänderungen, hinzugefügt werden.

8.4.3 DC-Teil

Im Duration Calculus Teil müssen wir uns deutlich einschränken, was die Formeln betrifft. Zugelassen sind nur sogenannte “counterexample”-Formeln. Sie haben stets folgende Struktur:

$$\neg\Diamond(phase_1 \,;\, \cdots \,;\, phase_i)$$

Sie werden benutzt um ein bestimmtes (zeitliches) Verhalten zu verbieten. Die Idee der Übersetzung orientiert sich daran, den abgebildeten Trace zu verbieten, indem ein Automat konstruiert wird, der in einem Zustand startet, der mit einer *true*-Phase korrespondiert. Diese kommt aus der Auflösung des $\Diamond$-Operators.
Die Konstruktion eines PEAs aus einer DC-Formel ist eine Potenzmengenkonstruktion. Jeder Zustand des PEAs wird mit einer Menge von Phasen beschriftet. Die Idee ist, dass der Zustand genau dann erreicht wird, wenn die in der Mengen enthaltenen Phasen gesehen wurden. Wir benötigen für jede Phase, die über Zeit redet, eine Uhr. Diese Uhr wird nach folgendem Prinzip zurück auf null gesetzt: Falls sie eine obere Schranke misst, dann so oft wie möglich, und falls sie eine untere Schranke misst, nur beim Neubetreten der Phase. Events dürfen nur auftreten, wenn in der Formel erwähnt. Das heißt, sie werden verboten, falls sie nicht gefordert werden. Transitionen führen wir so ein, dass wir von einer Phase in die nächste gelangen.

8.4.4 Model-Checking

Mit Hilfe der gemeinsamen Semantik für CSP-OZ-DC ist es möglich, Spezifikationen in dieser Kombination von Spezifikationssprachen zu erstellen und sie zu Model-Checken. Das Model-Checking geschieht auf folgendem Weg: Die Einzelteile der Spezifikation werden in PEA umgewandelt. Die parallele Komposition dieser PEA kann dann in *Timed Constraint Systems* übersetzt werden, für die Model-Checker bereits existieren. Die Übersetzung ist in [HM05] beschrieben.

8.4.5 Beispiel

Um einen Einblick zu bekommen, wie eine Spezifikation in CSP-OZ-DC und eine Übersetzung in PEA aussieht, betrachten wir folgendes Beispiel. Die CSP-OZ-DC-Spezifikation in Abbildung 8.2 beschreibt einen Zähler, der sich um 0 bis 2 erhöhen kann bei jedem Schritt, nie größer wird als 5 und bei dem 2 Resets mindestens 5 Zeiteinheiten auseinander liegen. Zusätzlich dürfen keine 5 Zeiteinheiten mehr vergehen, sobald der Zähler über 2 steigt, so dass dann auf jeden Fall ein Reset folgen muss. In Abbildung 8.3 ist der Automat zu sehen, der sich bei einer Übersetzung des CSP-Teils ergibt. Abbildung 8.4 zeigt den Automaten zum Object-Z-Teil. Die Übersetzung des DC-Teils ist in Abbildung 8.5 zu sehen.

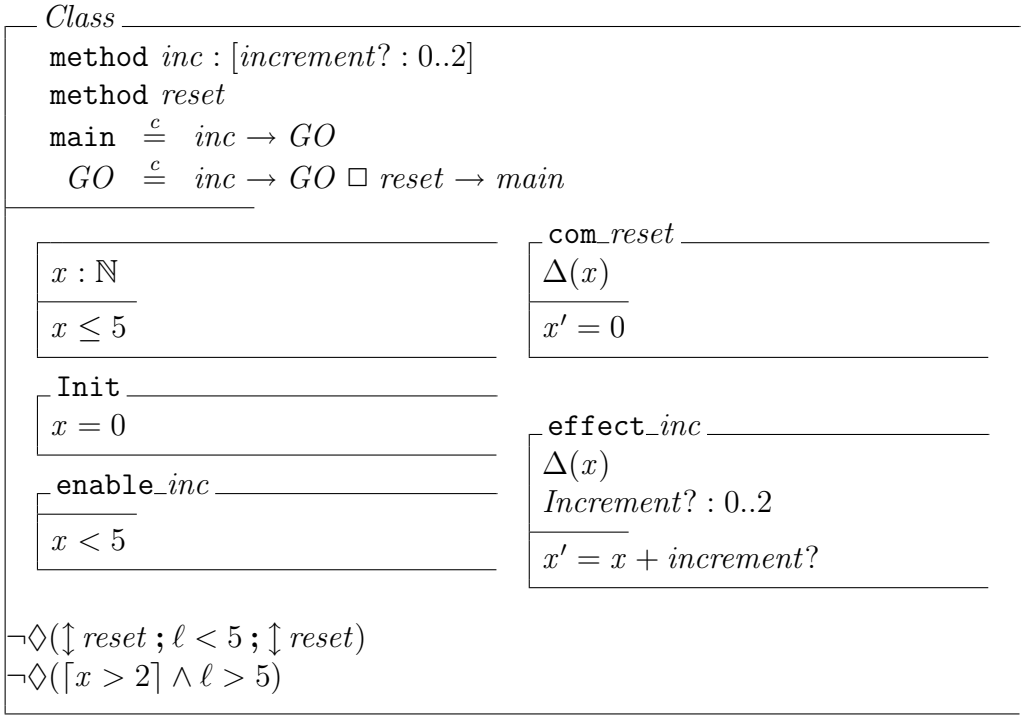


Abbildung 8.2: CSP-OZ-DC Spezifikation eines Zählers

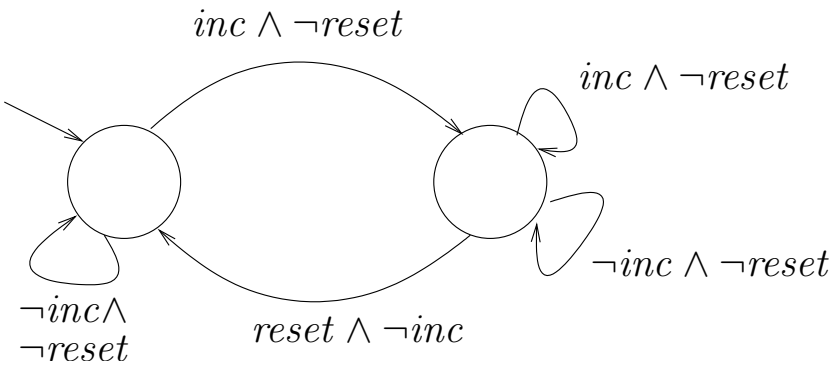


Abbildung 8.3: Automat zum CSP-Teil

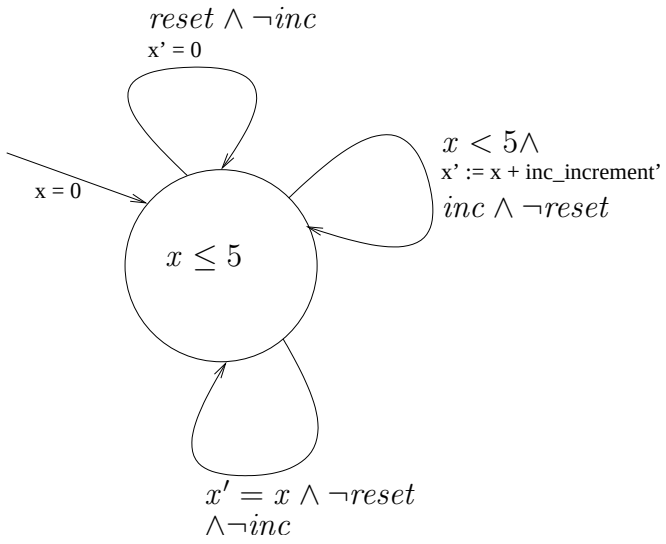


Abbildung 8.4: Automat zum Object-Z-Teil

8.5 Zusammenfassung

CSP-OZ ist eine Kombination von Spezifikationstechniken, mit der konfortables Modellieren möglich ist. Mit dem Hinzunehmen des Duration Calculus ist es auch möglich, zeitliche Zusammenhänge zu spezifizieren. Es existiert eine gemeinsame Semantik für CSP, Object-Z und den Duration Calculus in Phaseneventautomaten. Model-Checking der Spezifikationen ist möglich, indem eine Übersetzung in Phaseneventautomaten und von da aus in *Transition Constrain Systems* durchgeführt wird, für die Model-Checker existieren.

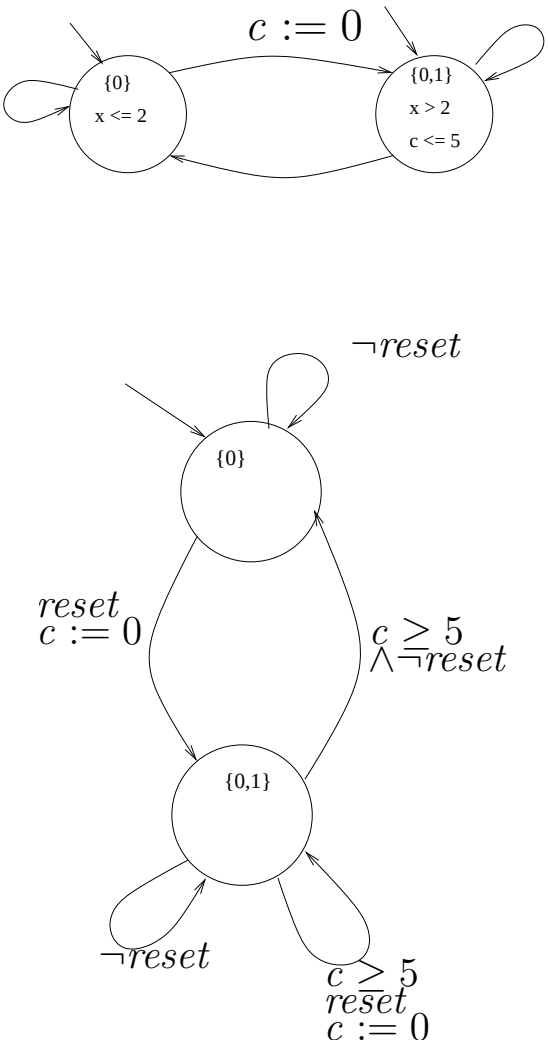


Abbildung 8.5: Automaten zum DC-Teil

Kapitel 9

Ein UML-Profil für CSP-OZ-DC

Sven Linker

9.1 Einleitung

In der Softwareentwicklung wird immer größerer Wert auf eine Strukturierung der Projekte mithilfe der UML gelegt. Da jedoch die UML keine exakte Semantik besitzt, sind die Diagramme häufig mehrdeutig, so dass sich die Korrektheit eines Designs nicht überprüfen lässt. Im Folgenden soll ein UML-Profil definiert werden, dass eine Übersetzung in die Systembeschreibungssprache CSP-OZ-DC ermöglicht. Hiermit wird auch den Diagrammen eine eindeutige Semantik hinzugefügt, so dass eine automatische Verifikation der entwickelten Systeme möglich wird.

9.2 UML-Profil

Zur grafischen Beschreibung eines Systems werden die aus der UML bekannten Klassendiagramme und Statemachines genutzt. Zusätzlich wurden aus der UML-RT[1] noch die sogenannten Strukturdiagramme übernommen. Im UML-Profil sind die Stereotypen *capsule*, *port* und *protocol* vorhanden. Als *capsules* werden die eigentlichen Systembausteine bezeichnet, die Operationen zu Verfügung stellen und nutzen. Diese Operationen sind jedoch nicht offen zugänglich, sondern werden über die *ports* anderen *capsules* zugänglich gemacht. Jeder *port* ist entweder als Sender (*base*, schwarz) oder als Empfänger (*conjugated*, weiß) definiert. Die Kommunikation zwischen zwei oder mehreren *ports* wird über die *protocols* eindeutig beschrieben. In diesen wird angegeben, welche Operationen über den entsprechenden Port zugänglich sind. Um festzulegen, in welcher *capsule* eine Operation implementiert wird, wird jeder Operation ein weiterer *stereotype* zugeordnet. Hierbei sind *in*,*out*,*in_S* und *out_S* erlaubt. *in* und *out* bezeichnen Eingangs-, bzw. Ausgangsoperationen, *in_S* und *out_S* die entsprechenden Broadcastoperationen.

[1] RT: Real-Time

Jede Klasse mit einem Stereotyp besitzt unterschiedliche *tags*, in die für die Übersetzung notwendige Spezifikationen eingetragen werden.

9.2.1 Capsule

Jede Klasse des Stereotyps *capsule* besitzt drei *tags*:

invariant definiert die Klasseninvariante

init beinhaltet die Initialisierungswerte der Klasse

dc counterexample enthält die für die Klasse geltenden DC-Gegenbeispiel-Formeln

invariant und *init* werden unmittelbar als OZ-Spezifikationen eingetragen, *dc counterexample* besteht, wie aus dem Namen ersichtlich ist, aus Gegenbeispielformeln des Duration Calculus.
Jede Operation der Klasse besitzt wiederum drei *tags*:

changes definiert die Attribute, die durch die Operation verändert werden

enable enthält die Vorbedingung für die Ausführung der Operation

effect definiert die Änderung der Attribute bei Ausführung der Operation

Diese *tags* werden alle mit OZ-Code gefüllt.

9.2.2 Protocol

Jedes Operation eines *protocols* besitzt die folgenden *tags*:

input die Eingabeparameter

output die Ausgabeparameter

simple die zur Adressierung einer Instanz nötigen Adressinformationen

Auch diese *tags* werden alle mit OZ-Code definiert.

9.2.3 Übersetzung der Klassendiagramme

Den größten Teil der Übersetzung macht die Umwandlung der Klassendiagramme aus. Hierbei werden normale Datenklassen in OZ-Klassen, *capsules* in CSP-OZ-DC-Klassen übersetzt. Die Attribute einer Klasse werden unmittelbar aus dem Diagramm übernommen, wobei bei einer Datenklasse alle Attribute in die *Visibility*-Liste eingetragen werden. Ist eine Datenklasse mit einer *capsule* assoziert, so wird diese Datenklasse bei der entsprechenden *capsule* als Attribut eingetragen (bei einer 1 zu n Beziehung als *finite set* $\mathbb{F}$).
Alle *tags* einer Klasse werden entsprechend ihrer Bezeichnung in die Klasse eingefügt, so wird zum Beispiel der Inhalt des *init-tags* in den Init-Block der Klasse kopiert.

Die Operationen eines *Protocols* werden in das Interface der assozierten *capsules* eingetragen. Dabei ist die Rolle des *ports* für die *capsule* entscheidend für die Übersetzung der Operationen. Für einen *base-port* werden die *in-* und *in_S*-Operationen zu `method`-Operationen, *out-* und *out_S* zu `chan`-Operationen. Bei der Übersetzung der *conjugated-ports* werden die Stereotypen der Operationen invertiert und anschließend analog umgewandelt. Auch die *tags* der Operationen werden entsprechend ihrer Bezeichnung in die Klasse eingetragen.

9.2.4 Übersetzung der Statemachines

Für die Übersetzung der Statemachines wird eine Übersetzungsabbildung φ benötigt. Hierfür werden erst einige Definitionen getroffen. Mit *SM* wird im Folgenden eine Statemachine, mit s einer ihrer Zustände bezeichnet. Die Menge aller CSP-Prozesse wird als *CSP* definiert. $\mathcal{C}_s$ ist die Menge aller direkten Nachfolger von s ohne *Triggerevent* und $\mathcal{T}_s$ ist die Menge aller Paare (e, t) der direkten Nachfolger t von s mit *Triggerevent* e. Weiterhin bezeichnet $\mathcal{I}_M$ die Menge der Initalzustände einer Untermaschine M von *SM*. Mittels dieser Definitionen wird nun die Abbildung $\varphi : SM \to CSP$ definiert:

$$\varphi \equiv \begin{cases} P_s = SKIP & \text{wenn } s \text{ Endzustand ist} \\ P_s = \Box_{(e,t)\in\mathcal{T}_s} e \to P_t & \text{wenn } s \text{ einfacher Zustand ist und } \mathcal{C}_s = \varnothing \\ P_s = \bigsqcap_{t\in\mathcal{C}_s} P_t & \text{wenn } s \text{ einfacher Zustand ist und } \mathcal{C}_s \neq \varnothing \\ & \text{oder wenn } s \text{ Initialzustand ist} \\ P_s = (\ |||_{i=1}^{n} P_{M_i}); & \\ \textbf{if}\, \mathcal{C}_s \neq \varnothing\, \textbf{then}\, (\bigsqcap_{t\in\mathcal{C}_s} P_t)\, \textbf{else}\, STOP & \text{wenn } s \text{ zusammengesetzter Zustand ist} \\ & \text{mit Untermaschinen } M_i, 1 \leq i \leq n. \end{cases}$$

$\varphi(M) \equiv P_M = \bigsqcap_{t\in\mathcal{I}_M} P_t$ für jede Untermaschine M von *SM*.

Diese Abbildung wird auf die gesamte Statemachine und alle ihre Untermaschinen angesetzt und das Ergebnis als CSP-Spezifikation in die zur Statemachine gehörige Klasse eingetragen ([RW03]).

9.2.5 Übersetzung der Strukturdiagramme

Bei der Übersetzung der Strukturdiagramme wird zunächst für jede Instanz einer *capsule* ein Prozess parallel erzeugt. Damit die Kommunikation über die *ports* möglich ist, müssen die Prozesse auf den möglichen Operationen synchronisiert laufen. Das heißt, dass Klassen, die über einen *port* verbunden sind, über die im dazugehörigen *protocol* definierten Operationen synchronisiert werden müssen. Die Vereinigungsmenge aller Operationen der *ports* zwischen zwei Klassen ist die Synchronisationsmenge der Prozesse.
Eine Besonderheit ergibt sich bei den Broadcast-Operationen. Aufgrund der Spezifikation von CSP ist es nötig, dass alle Prozesse, die über eine solche Operation angesprochen werden, auch auf dieser synchronisiert werden müssen. Werden also drei Instanzen einer Klasse über eine Broadcast-Operation angesprochen, müssen sie auch auf dieser

synchronisiert werden.
Alle Operationen, die innerhalb des Strukturdiagramms (also innerhalb der entsprechenden *capsule* oder des gesamten Systems) genutzt werden, müssen als `local_chan` definiert werden. Nur Operationen, die über einen *port* nach außen durchgeschleift werden, werden wie bei den Klassendiagrammen in die entsprechende CSP-OZ-DC-Klasse eingetragen ([FOW01]).

9.3 Beispiel

Als Beispiel dient eine einfache Bankverwaltung (entnommen aus [MORW05]). Diese besteht aus einer Bank und drei Geldautomaten, den ATMs (*A*utomatic *T*eller *M*achine). Auf eine exakte Erläuterung des Beispiels wird aufgrund Platzmangels verzichtet. Weiterhin werden bei der Übersetzung des Beispiels die *tags*, die im Diagramm nicht zu erkennen sind, nicht mit in die Klassen eingetragen, um die Übersichtlichkeit zu erhöhen. Wie in Abbildung 9.1 zu sehen ist, besitzt die Bank zwei *ports* (im

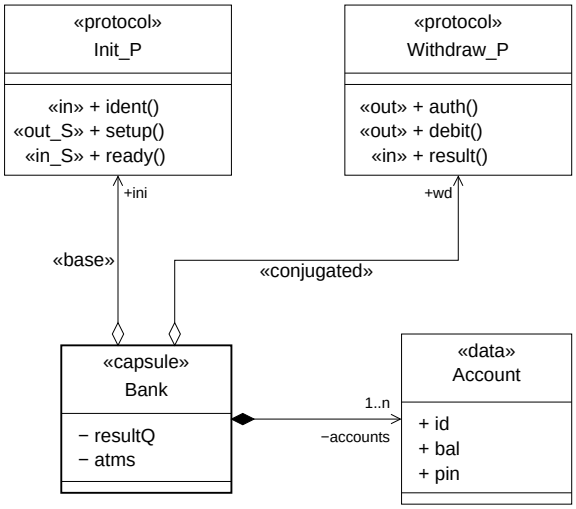


Abbildung 9.1: Klassendiagramm der *Bank* inklusive der benötigten Protokolle (aus [MORW05])

Diagramm als Assoziationen zu erkennen), die eine Kommunikation mit den ATMs ermöglichen. Führt man eine Übersetzung nach dem oben angegebenen Schema durch, so erhält man für die Klasse *Account*:

Account
$\lceil(id, bal, pin)$

id
bal
pin

Die Klasse *Bank* hingegen ist etwas komplexer.

Bank

`method` *ident*
`method` *ready*
`method` *auth*
`method` *debit*
`chan` *setup*
`chan` *result*

$accounts : \mathbb{F} Account$
$resultQ$
$atms$

`Init`

Wie zu sehen ist, besteht die Klasse nur aus dem Interface und den OZ-Definitionen. Für die CSP-Spezifikation muss die Statemachine (Abbildung 9.2) übersetzt werden: Mittels der Übersetzungsrelation φ und einigen Vereinfachungen ergibt sich hieraus

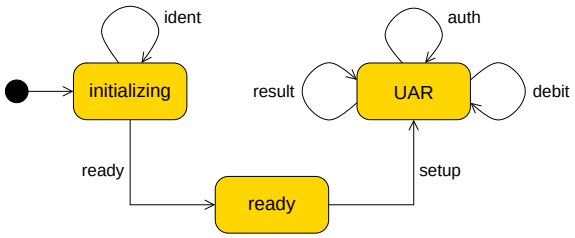


Abbildung 9.2: Statemachine der *Bank* (aus [MORW05])

$$main = P_M = ident \rightarrow P_M \ \square \ ready \rightarrow setup \rightarrow P_{UAR}$$
$$P_{UAR} = result \rightarrow P_{UAR} \ \square \ auth \rightarrow P_{UAR} \ \square \ debit \rightarrow P_{UAR}$$

Abschließend muss noch das Strukturdiagramm der Anwendung (Abbildung 9.3) übersetzt werden. Hier ist zu sehen, dass eine Instanz der *Bank*, zwei *ATM*s und eine *ML_ATM* (*M*ulti-*L*anguage ATM) zu erzeugen sind. Die Bank und die ATMs müssen über den gesamten Operationen, die die Bank zur Verfügung stellt, synchronisiert werden. Das Synchronisationsalphabet der ATMs besteht aus den Broadcast-Operationen *setup* und *ready*. Werden nun noch die Operationen in die Klasse eingetragen, so erhält man:

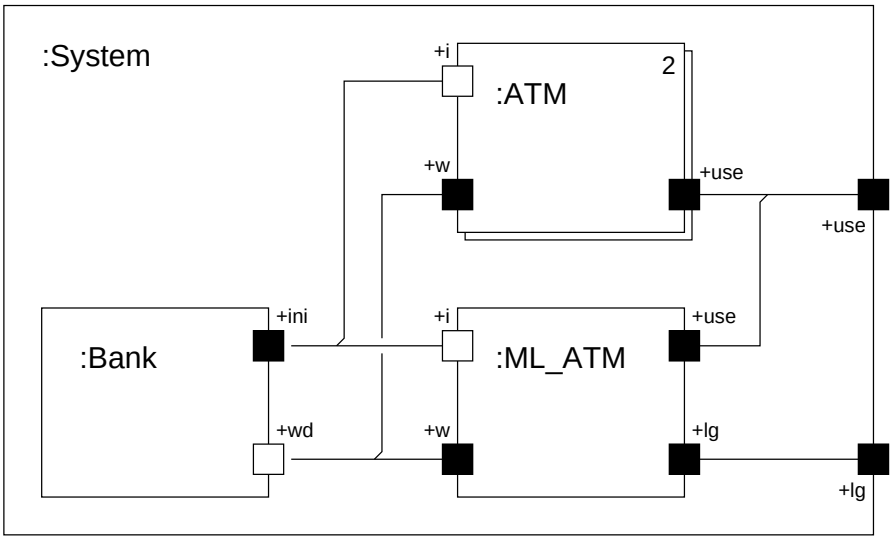


Abbildung 9.3: Strukturdiagramm der Bankverwaltung (aus [MORW05])

System

`local_chan` *ident*
`local_chan` *ready*
`local_chan` *setup*
`local_chan` *auth*
`local_chan` *debit*
`local_chan` *result*

$$\texttt{main} = Bank \underset{\{ident,ready,auth,debit,setup,result\}}{\parallel} CLIENTS$$

$$CLIENTS = (ATM \underset{\{ready,setup\}}{\parallel} ATM \underset{\{ready,setup\}}{\parallel} ML_ATM)$$

Hierbei wurden die *ports* `use` und `lg` ignoriert, um die Übersichtlichkeit zu erhöhen, und um keine Operationen zu definieren, auf die im gesamten restlichen Beispiel keine Rücksicht genommen wurde.

9.4 Zusammenfassung

Für das UML-Profil mussten einige Einschränkungen getroffen werden. So müssen *capsules*, die von einer anderen *capsule* erben, deren Statemachine implementieren. Die einzige erlaubte Erweiterung ist die parallele Ausführung einer neuen Statemachine. Weiterhin ist zu beachten, dass ein *base-port* nur mit einem *conjugated-port* verbunden werden darf und nicht mit einem weitern *base-port*. Eine Ausnahme hiervon bildet das Delegieren einer Portfunktion von einer äußeren *capsule* auf eine innere.
Mithilfe dieses UML-Profils wurde gezeigt, dass eine eindeutige Übersetzung von UML in CSP-OZ-DC möglich ist. Damit wurde gleichzeitig der UML eine definierte Semantik hinzugefügt.

Kapitel 10

Software testen

Christian Wenzel

10.1 Einleitung

Dieses Dokument ist im Rahmen der Projektgruppe Syspect an der Universität Oldenburg entstanden. Es erläutert eine Vorgehensweise beim Testen von Software. Durch das Testen von Software soll das tatsächliche Verhalten dieser mittels Testobjekte und eines Testplans untersucht werden. Der Begriff Testobjekt bezeichnet konkrete Instanzen von Klassen, die zum Testen des Systemcodes verwendet werden. Die Ergebnisse dieser Untersuchung werden mit den Soll-Ergebnissen (laut Anforderung) verglichen und dokumentiert. Das Testen sollte den gesamten Entwicklungsprozess begleiten und nicht am Ende des Projektes unter Zeitdruck durchgeführt werden. Im Folgenden wird zuerst eine Klassifizierung der einzelnen Teststufen gegeben. Insbesondere wird auf das Testen von Komponenten mit JUnit unter der Java-Technologie eingegangen. In diesem Zusammenhang wird auch das Entwicklungswerkzeug ANT erläutert. Des Weiteren wird die *Java Modeling Language* (JML), die zur Spezifikation von Java-Programmen dient, vorgestellt.

10.2 Teststufen

In diesem Kapitel wird eine Einteilung der Softwaretests in verschiedene Stufen vorgestellt.

- **Komponententests**
  Bei Komponententests werden einzelne Einheiten (Klassen bzw. Methoden dieser) individuell und unabhängig von anderen Einheiten des Systems getesten. Dies kann gegen den Programmcode (White-Box-Test) oder die Spezifikation (Black-Box-Test) erfolgen.

- **Integrationstests**
  Das Testen einzelner Komponenten garantiert das korrekte Zusammenwirken dieser nicht. Einige Module können nicht isoliert getestet werden. Aus diesem Grunde sollten integrative Tests verschiedener Teilbereiche durchgeführt werden.

- **Systemtests**
  Abschließend wird die Funktionalität, die Zuverlässigkeit, die Effizienz und die Benutzbarkeit des gesamten Systems getestet. In diesem Zusammenhang wird überprüft, ob alle funktionalen und nicht funktionalen Anforderungen erfüllt sind.

- **Abnahmetests**
  Dieser Test ist eine besondere Art des Systemtests. Der Auftraggeber überprüft, ob das entwickelte System die vertraglichen Vereinbarungen erfüllt, d.h. das System wird gegen das Pflichtenheft, das den vertraglichen Rahmen vorgibt, geprüft.

Die einzelnen Teststufen bauen hinsichtlich der Testobjekte aufeinander auf. Das Testen von höheren Stufen sollte erst dann erfolgen, wenn die Tests niedriger Stufe mit ausreichendem Erfolg abgeschlossen wurden. Die Gesamtheit aller Tests lässt eine Bewertung der Qualität der Software zu und ist somit ein wesentlicher Bestandteil der Qualitätssicherung [Sch04].

10.3 JUnit

JUnit ist ein Framework, das zum Testen von Java-Programmen entwickelt wurde. Die beiden Hauptverantwortlichen dieses Projektes waren Erich Gamma und Kent Beck. JUnit ist Open-Source-Software und wurde unter der *IBM Public License* veröffentlicht. Mittlerweile existieren ähnliche Produkte für andere Programmiersprachen. Besonders geeignet ist dieses Testframework zum Automatisieren von Tests bezüglich einzelner Klassen. Im Zusammenhang mit JUnit wird häufig vom so genanntem *Test Driven Development* (TDD) gesprochen. Dies bedeutet, dass das Testen ein kontinuierlicher Prozess ist, der ebenfalls einen Bestandteil der eigentlichen Softwareentwicklung darstellt. Beim TDD wird zuerst der Testcode erzeugt und dann der eigentliche Programmcode, der die Testbedingungen erfüllen soll. Diese Methodik ist in der Softwareentwicklung besonders für „agile“ Vorgehensmodelle, wie z.B. *Extreme Programming*, geeignet.

10.3.1 JUnit-Framework

Zum Testen von Software werden eigene Testklassen erstellt. Diese sind von der abstrakten Klasse `TestCase` aus dem Package `junit.framework` abgeleitet. Die Klasse `TestCase` erweitert wiederum die Klasse `Assert`.
In diesem Abschnitt werden die Bestandteile von JUnit näher vorgestellt, die zur Entwicklung von Komponententests benötigt werden.

Assert

Mit der Klasse `Assert` des Testframeworks stehen dem Entwickler Methoden zur Verfügung, um Werte oder Bedingungen zu testen. Der Zugriff auf diese Methoden erfolgt

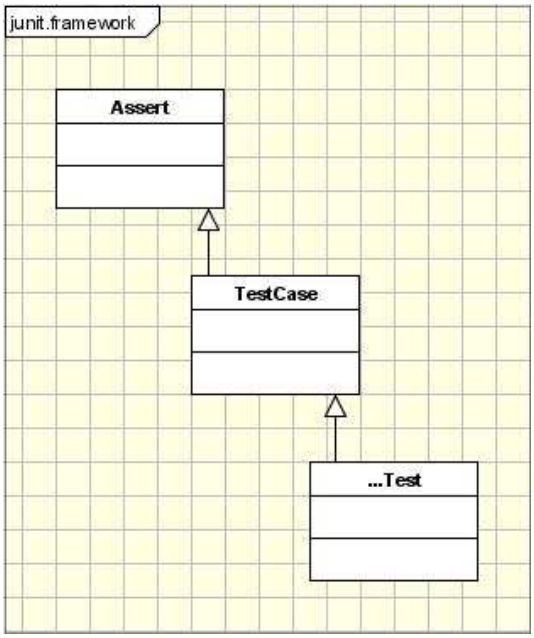


Abbildung 10.1: Klassenstruktur unter JUnit

über Vererbung (s.o.). Die folgende Aufzählung zeigt einen Überblick über die wichtigen Methoden:

- `assertTrue(boolean condition)`
  Diese Methode verifiziert, ob eine Bedingung erfüllt wurde.

- `assertEquals(Object o1, Object o2)`
  Diese Methode verifiziert, ob zwei Objekte gleich sind. Die Überprüfung geschieht mittels der `equals`-Methode. Diese Methode ist auch für primitive Datentypen, wie z.B. `int`, `double` oder `boolean`, definiert.

- assertNull(Object o)
  Diese Methode verifiziert, ob die Objektreferenz null ist.

- `assertNotNull(Object o)`
  Diese Methode verifiziert, ob die Objektreferenz nicht null ist.

- `assertSame(Object o1, Object o2)`
  Diese Methode verifiziert, ob die beiden Referenzen auf den gleichen Speicherbereich zeigen.

Wenn die durch `assert`-Methoden kodierten Behauptungen einen Fehler verursachen, wird der Testfall sofort durch das Framework abgebrochen und eine `AssertionFailedErrorException` wird geworfen. Neben den auftretenden Fehlern der aufgestellten Behauptungen (*Failure*) können auch, wie bei allen anderen Java-Programmen, Fehler in der Programmlogik (*Error*) des Tests auftreten.

TestCase

Die Klasse `TestCase` muss erweitert werden, um Testklassen mit dem JUnit-Framework zu erstellen. Diese Testklassen enthalten einzelne Testfälle in Form von Methoden. Mit einem Testfall resp. der Methode soll die entsprechende Methode der zu testenden Klasse überprüft werden. Die Testmethoden müssen folgenden Konventionen entsprechen, um ausgeführt werden zu können:

- Der Methodenname beginnt mit `test`.
- Die Methode ist `public` und nicht `static`.
- Die Methode besitzt keine Parameter und keinen Rückgabewert.

Das folgende Listing zeigt einen beispielhaften Aufbau einer Testklasse.

```
import junit.framework.*;
public class IntTest extends TestCase {

     public IntTest(String name) {
          super(name);
     }

     public void testAdd() {
          Int i = new Int(4);
          Int expected = new Int(7);
          Int result = i.add(3);
          assertEquals(result, expected);
     }
}
```

Listing 1: Einfacher Testfall

Neben den Testmethoden muss die Testklasse einen Konstruktor haben, der als Übergabeparameter eine Zeichenkette besitzt. Dieser ruft den Konstruktor der Oberklasse auf. Das Beispiel zeigt die Testmethode `testAdd`, die zum Testen der entsprechenden Methode `add` der Klasse `Int` benutzt wird. Das Ergebnis wird mit dem erwarteten Wert über die Methode `assertEquals` verglichen. Testmethoden sollen um Testobjekte gruppiert werden, d.h. Testfälle, die gegen die gleiche Menge von Testobjekten getestet werden, werden unter dem Dach einer Testklasse zusammengefasst. Diese Menge von Objekten wird auch als *Test-Fixtures* bezeichnet. Wenn eine Testmethode die *Test-Fixtures* nicht benutzt, ist das ein Anzeichen dafür, dass diese in einer anderen Testklasse implementiert werden sollte. Aus diesem Grunde ist es möglich, dass zu einer Klasse mehrere Testklassen existieren. Damit einzelne Testfälle isoliert voneinander ablaufen können, werden die *Test-Fixtures* für jeden Testfall neu initialisiert. Deswegen werden die Objekte als Instanzvariablen der Testklasse dargestellt. Zum Initialisieren wird die Methode `setUp` der Klasse `TestCase` überschrieben. Um nach Ablauf des Testfalls Ressourcen, wie z.B. Datenbankverbindungen, wieder freizugeben, wird dementsprechend die Methode `tearDown` überschrieben.

Lebenszyklus von Testfällen

Beim Ausführen von jedem Testfall durch das JUnit-Framework wird ein Zyklus durchlaufen. Vor diesem müssen die einzelnen Testfälle ermittelt werden. Dies erfolgt durch die *Reflection API* von Java. Diese ermöglicht es Klassen bzw. Objekten, die zur Laufzeit von der *Java Virtual Machine* im Speicher gehalten werden, auf Merkmale zu untersuchen. Die einzelnen Testmethoden werden durch die Zeichenkette `test` (s.o.) am Anfang identifiziert. Die einzelnen Testfälle werden unsortiert gesammelt und nacheinander ausgeführt.
Der Lebenszyklus eines Testfalls sieht dann wie folgt aus: Um Seiteneffekte zu vermeiden, wird für jeden Testfall eine neue Instanz der Testklasse durch JUnit erzeugt. Vor dem Ablauf der Testmethode wird die `setUp`-Methode aufgerufen, um die *Test-Fixtures* zu initialisieren. Danach wird die `test...`-Methode ausgeführt und abschließend die `tearDown`-Methode. Die Methoden `setUp` bzw. `tearDown` werden nur aufgerufen, wenn sie in den Testklassen überschrieben worden sind.

Testsuiten

Um das Testen des Programmcodes zu automatisieren, können die einzelnen Testklassen zu einer Testsuite zusammengefasst werden. Dies soll das Ausführen aller Testfälle „in einem Rutsch“ ermöglichen, um unnötigen Testaufwand zu vermeiden. Das folgende Listing zeigt den programmtechnischen Aufbau einer Testsuite.

```
import junit.framework.*;
public class AllTests {

     public static Test suite() {
          TestSuite suite = new TestSuite();
          suite.addTestSuite(IntTest.class);
          ...
          return suite;
     }
}
```

Listing 2: Testsuite erstellen

Zum Erzeugen einer Testsuite wird eine Klasse implementiert, die die statische Methode `suite` besitzt. Die eigentliche Testsuite wird durch eine Instanz der Klasse `TestSuite` dargestellt. Dieser werden die einzelnen Testklassen hinzugefügt. Die Klasse `TestSuite` implementiert, wie auch die Klasse `TestCase`, das Interface `Test`. Dieses Entwurfmuster, das unter dem Namen Komposition bekannt ist [GHJ95], ermöglicht beliebig viele Testklassen und Testsuiten zu einer gesamten Testsuite zusammenzufassen.

Test-Runner

In den bisherigen Abschnitten dieses Kapitels wurde darüber gesprochen, wie Testfälle erzeugt und zu Testsuiten zusammengefasst werden können. Dieser Abschnitt stellt

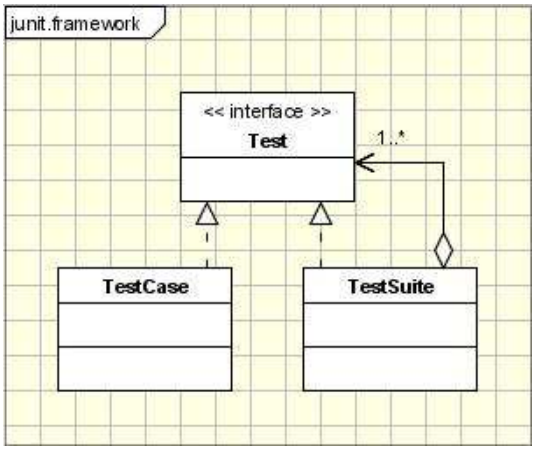


Abbildung 10.2: Klassenstruktur unter JUnit

kurz die Ausführung der Testfälle bzw. -suiten dar. JUnit bietet die Möglichkeit, die Tests über eine `main`-Methode in einer Klasse auszuführen oder allgemein über einen Kommandozeilenaufruf. Weiterhin können die Ergebnisse der Tests ebenfalls in einem Kommandozeilenfenster ausgegeben oder in einem grafischen Dialog präsentiert werden. Zusätzlich zu der einfachen Ausführung der Tests können diese automatisch wiederholt werden. Diese Option kann beispielsweise benutzt werden, um Testparameter aus einer Datei zu lesen [Wes05].

10.3.2 ANT: JUnit-Task

Dieser Abschnitt stellt eine alternative Möglichkeit vor, Testsuiten mit dem Entwicklungswerkzeug ANT (*Another Neat Tool*) zusammenzustellen. Das Build-Werkzeug ANT ist eine Java-Technologie, die zum automatischen Erstellen von Softwarepaketen entwickelt wurde. Die Konfiguration erfolgt über eine XML-Datei, die von ANT ausgewertet wird. Der grundsätzliche Aufbau einer Konfigurationsdatei besteht aus der Definition eines Projektes, das einzelne Ziele (*targets*) definiert. Diese Ziele wiederum beinhalten Aufgaben (*tasks*), die bei Aufruf des Ziels ausgeführt werden. Solche Aufgaben sind u.a. das Kompilieren von Quellcode oder das Übertragen von Daten zu einem Server. Daneben kann man noch Abhängigkeiten zwischen einzelnen Zielen definieren. Für weitere Informationen sei auf die einschlägige Literatur verwiesen. Unter ANT gibt es neben den o.ä. Aufgaben auch die Möglichkeit ein JUnit-*Task* zu definieren [Lem06]. Das folgende Listing zeigt den Aufbau dieses Tasks:

```
<?xml version="1.0"?>
<project name="JUnit demo" basedir=".">
     <target name="junit" depends="compiletest">
          <junit>
               <formatter type="plain" usefile="false"/>
               <!-- beliebig viele -->
               <test name="IntTest"/>
```

```
            </junit>
        </target>
    </project>
```

Listing 3: JUnit-Task unter ANT

Am Anfang können über das Tag `formatter` einige Parameter gesetzt werden. Das Attribut `type` konfiguriert die Protokollierung der Tests und das Attribut `usefile` ermöglicht die Ausgabe in eine Datei. Gefolgt von der Konfiguration können beliebig viele Testfälle bzw. -suiten über das Tag `test` hinzugefügt werden. Bevor die einzelnen Tests ausgeführt werden können, müssen die Dateien kompiliert werden. Dies wird mittels der Abhängigkeiten (`depends`) von einzelnen Zielen erreicht [JNT06].

10.4 Java Modeling Language

Die *Java Modeling Language* (JML) ist eine programmiernahe Spezifikationssprache für Java. Sie beschreibt das Laufzeitverhalten von Programmen in Form von Zusicherungen (*Design by Contract*). Diese Verträge werden direkt im Quellcode definiert. Ein spezieller Compiler (*jmlc*) kann diese Spezifikationen in ausführbaren Code übersetzen, der zur Laufzeit die Bedingungen überprüft. Des Weiteren kann JML auch als Basis für diverse Verifikationstools, wie z.B. LOOP oder ESC-Java, benutzt werden [JML06].

10.4.1 Syntax

JML-Zusicherungen werden in Form von Kommentaren in den Quellcode eingebettet. Diese speziellen Kommentare werden durch ein @-Zeichen am Anfang gekennzeichnet. Der Java-Compiler behandelt alle JML-Spezifikationen als einfache Kommentare. Diese Kompatibilität ermöglicht die Verwendung der JML-Dateien auch als Java-Dateien. Im Folgenden werden wichtige Sprachelemente der Spezifikation zusammengefasst.

Verhaltensbeschreibung von Methoden
Eine Verhaltensbeschreibung wird durch das Schlüsselwort `behavior` eingeleitet und steht vor der zu beschreibenden Methode. Für eine genauere Beschreibung des Verhaltens existieren die beiden Varianten `normal_behavior` für ein normales Verhalten und `exceptional_behavior` für die Beschreibung im Ausnahmefall. Unterschiedliche Verhaltensbeschreibungen, z.B. bei verschiedenen Vorbedingungen, sind ebenfalls möglich. Die korrespondierenden Beschreibungen sind in diesem Fall durch das Schlüsselwort `also` zu verbinden. Vor- und Nachbedingungen von Methoden werden durch die Schlüsselwörter `requires` bzw. `ensures` eingeleitet. Diesen Schlüsselwörtern folgt ein logischer Ausdruck. Wenn eine Methode eine `exception` als Ergebnis liefert, wird das durch das Schlüsselwort `signals` beschrieben. Wenn Instanzattribute durch eine Methode verändert werden sollen, müssen sie in der Verhaltensbeschreibung angegeben werden. Diese Angabe wird durch das Schlüsselwort `assignable` gekennzeichnet. JML erlaubt in Methoden, sogenannte *Checkpoints* mittels `assert` zu definieren.

Invarianten
Invarianten müssen vor und nach jedem Methodenaufruf gelten, d.h. sie gelten zu jedem stabilen Zeitpunkt. Sie werden durch das Schlüsselwort `invariant` gekennzeichnet.

Modellvariablen
Modellvariablen sind eigens für die Spezifikation eingeführte Variablen. Die Deklaration wird durch das Schlüsselwort `ghost` und die Initialisierung wird mit dem Schlüsselwort `set` eingeleitet. Des Weiteren gibt es Modellvariablen, die mit dem Schlüsselwort `model` eingeleitet werden. Die Verbindung zum Programmcode wird mit Hilfe einer Abstraktionsfunktion (Schlüsselwort `represents`) hergestellt.

Quantoren
Für prädikatenlogische Ausdrücke können die Quantoren `\forall` und `\exists` benutzt werden. Diese haben folgende Form:

```
(<Quantor> <VariablenDeklarationen>;
 <Variablenbeschreibungen>;
 <Quantorenbedingung>)
```

Prädikate
Unter der JML-Spezifikation existieren verschiedene Prädikate. Die beiden wichtigsten sind `\old` und `\result`. Über `\old(...)` ist es möglich, auf Werte einer Variablen zu Beginn der Methode zu zugreifen. `\result` bezieht sich auf den Rückgabewert der Methode.

Beispiel

```
public class MathOps {
     private int a;
     //@ private invariant a  >  2
     MathOps(int a) { this.a =a;}
     /* @ public normal_behavior
        @  requires y >= 0;
        @  ensures \result <= y;
        @ also public exceptional_behavior
        @ requires y<0;
        @  signals (Exception e);
        @*/
     public static int getSqrt(int y) {
          if (y<0) throw new Exception();
          return (int) Math.sqrt(y);
     }
}
```

Listing 4: Einfaches JML Beispiel

Das obere Beispiel zeigt eine Klasse `MathOps` mit einer Methode `getSqrt`, die die Wurzel des Übergabeparameters zurückgibt. Für die Instanzvariable wird mit Hilfe der JML-Spezifikation eine Invariante definiert. Die Methode besitzt Spezifikationen für das normale Verhalten und das Verhalten in Ausnahmesituationen. Die letzte besagt, dass bei einer bestimmten Vorbedingung (y<0) eine Ausnahme geworfen wird [FH06].

10.4.2 Fehlerbehandlung

Wenn die aufgestellten Zusicherungen der JML-Spezifikation nicht erfüllt werden, werden entsprechende Ausnahmen geworfen. Verletzungen von Vorbedingungen lösen einen `JMLEntryPreconditionError` oder `JMLInternalPreconditionError` aus. Der letztere wird ausgelöst, wenn ein interner Aufruf einer anderen Methode eine Ausnahme auslöst. Dementsprechend gibt es die Ausnahme `JMLPostconditionError` für Verletzungen von Nachbedingungen. Wenn die Bedingung einer Invarianten nicht mehr eingehalten wird, löst dies einen `JMLInvariantError` aus.

10.4.3 jmlunit

jmlunit ist ein Werkzeug, dass aus Java-Klassen mit JML-Spezifikation Testklassen auf Basis von JUnit generiert. In diesem Zusammenhang spricht man auch von *Model-Driven-Generation*: Ein Generator erhält als Eingabe eine abstrakte Beschreibung durch eine Spezifikation. Durch dieses Vorgehen wird die Entwicklung der Tests automatisiert, da der eigentliche Testcode nicht mehr durch den Entwickler geschrieben werden muss. Die folgende Grafik zeigt den Ablauf der Generierung:

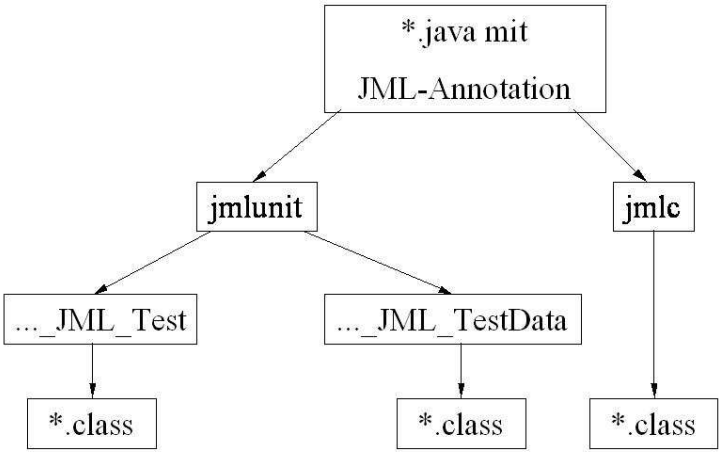


Abbildung 10.3: Generierung von JUnit-Testklassen aus JML-Dateien

Aus einer Java-Klasse mit JML-Spezifikationen werden zwei Testklassen generiert. Zum einen die Datei `..._JML_TestData`, die von der Klasse `TestCase` abgeleitet ist. Die Klasse `..._JML_Test` ist von der Klasse `..._JML_TestData` abgeleitet. Sie enthält die generierten Testmethoden aus der zu testenden Klasse. Des Weiteren enthält sie Standardtestwerte. Die JML-Spezifikation dient hier als so genanntes „Test-Orakel“, d.h. sie

ist die Informationsquelle zur Ermittlung von Sollergebnissen der Testfälle (Spezialfälle der Spezifikation). Neben den generierten Testdaten kann man eigene Testdaten hinzufügen. Diese sollte man in die Datei `..._JML_TestData` eintragen, da diese nur einmal erzeugt wird. Diese Testklassen werden schließlich kompiliert und testen die durch den jmlc-Compiler generierte Klasse [YC06].

10.5 Zusammenfassung und Beurteilung

Mit JUnit ist es schnell und einfach möglich, Testszenarien für ein gesamtes Java-Projekt aufzubauen. Dies wird vor allem dadurch gewährleistet, dass einzelne Testklassen sich zu verschiedenen Testsuiten kombinieren lassen, um komplexe Testhierarchien entwickeln zu können. Nachteilig ist zu bemerken, dass für dieses Werkzeug die einzelnen Testklassen vom Entwickler geschrieben werden müssen. Dieses Problem erhöht den Aufwand erheblich. Abhilfe könnte die Spezifikation mit JML bringen. Diese hat den Vorteil, dass der normale Java-Compiler sie als Kommentare behandelt und somit Kompatibilität gewährleistet ist. Das Werkzeug *jmlunit* ermöglicht es, mit Hilfe von JML-Spezifikationen JUnit-Testklassen zu erzeugen. Dieses entkräftet das oben genannte Problem der Mehrarbeit.

Kapitel 11

Versionsverwaltung

Dominik Denker

11.1 Einleitung

Im Rahmen dieser Ausarbeitung wird darauf eingegangen, was eigentlich Versionsverwaltung ist, z.B. was für Ansätze es gibt, und wie man die beiden Programme, CVS [FB05] und Subversion [CSFP05, Doh03], benutzt bzw. administriert. Nachdem ich dann kurz auf verschiedene GUIs eingegangen bin, werde ich die Versionsverwaltungsprogramme CVS und Subversion einem Vergleich unterziehen und ein Fazit bilden.

11.1.1 Versionsverwaltung

> Unter einer Versionsverwaltung versteht man ein System, welches typischerweise in der Softwareentwicklung zur Versionierung und um den gemeinsamen Zugriff auf Quelltexte zu kontrollieren, eingesetzt wird. Hierzu werden alle laufenden Änderungen erfasst und alle Versionsstände der Dateien in einem Archiv mit Zeitstempel und Benutzerkennung gesichert. [Wik05a]

Ersteinmal ein paar Worterklärungen:
Ein Repository ist eine Art Lager, das lokal oder auch weit entfernt Daten lagert, und im Falle des Versionsmanagements Änderungen an Dateien speichert. Mit Aus- und Einchecken ist gemeint, dass man Daten aus dem Repository in sein Arbeitsverzeichnis ausliest, bzw. von seinem Arbeitsverzeichnis in das Repository lädt. Mergen bezeichnet das Zusammenführen unterschiedlicher Dateiversionen.
Es existieren mindestens zwei Ansätze der Versionsverwaltung. Das *Copy-Modify-Merge*-Prinzip und das *Lock-Modify-Write*-Prinzip, wobei ersteres der Philosophie der *Optimistic Revision Control* folgt und zweiteres der der *Pessimistic Revision Control*. Sowohl CVS als auch Subversion bedienen sich ersteren Konzepts.
Mit *Copy-Modify-Merge* ist gemeint, dass man sich die Daten aus einem Repository

ausliest, diese ändert und dann wieder eincheckt. Sollte es bei einem Update ein Problem geben, werden beide Versionen der Datei so weit wie möglich zusammengesetzt. Sind von den Benutzern nur unterschiedliche Zeilen geändert worden, wird alles ohne Probleme zusammengefügt. Gibt es einen Konflikt wird dies angemerkt und die Datei so modifiziert das beide Änderungen vorhanden sind. In diesem Fall müssen die Änderungen von Hand, am Besten nach Kommunikation mit der anderen Person, gemerged und dann eingecheckt werden. Wie dies von CVS und Subversion gelöst wird, wird in 11.2.4 erklärt.
Beim pessimistischen Ansatz werden Dateien, die gerade bearbeitet werden, gesperrt und sind somit für niemand anderen zugänglich, bis diese wieder freigeschaltet werden. Das schließt Dateikonflikte zwar aus, behindert aber meist die Produktivität.

11.1.2 Versionierungsansatz

Auch wenn sie der gleichen Philosophie folgen, haben CVS und Subversion doch unterschiedliche Versionierungsansätze.
CVS vergibt für jede Datei einzelne Versionsnummern. Das heißt es existiert zum Beispiel die „Revision Fünf der Datei test.txt“ und daneben die „Revision Eins der Datei tut.txt“. Bei Subversion wird die Versionsnummer dem ganzem Repository gegeben. Das heißt, wenn die fünfte Version der Datei test.txt im Repository liegt, hat auch das Repository die Versionsnummer Fünf. Wenn dann die Datei tut.txt dazukommt, hat das ganze Repository die Versionsnummer Sechs. Die Datei tut.txt hat ebenfalls die Versionsnummer Sechs, allerdings exisitieren keine Versionen Eins bis Fünf. Die Datei test.txt hat weiterhin die Versionsnummer Fünf.

11.2 Grundlegende Operationen

Dieses Kapitel beschreibt vier der grundlegendsten Operationen näher, die sich in beiden Programmen in der Syntax kaum unterscheiden, aber doch teilweise unterschiedlich arbeiten. Teilweise existieren für diese Befehle auch Abkürzungen oder Synonyme, auf die aber nicht näher eingegangen wird.

11.2.1 Auschecken

- svn checkout [Optionen] [URL] [Pfad]
- cvs checkout [Optionen] [Projekte]

Um bei einem versionsverwalteten Projekt mitarbeiten zu können, das mit CVS oder Subversion versioniert wird, muss man sich als erstes eine lokale Arbeitskopie erstellen. Dies geschieht mit dem Befehl `checkout`. Wie man sieht, unterscheidet sich die Syntax in beiden Programmen nur wenig, allerdings arbeiten sie unterschiedlich.
Während CVS einfach alle Dateien und Verzeichnisse des angegebenen Projektes aus dem Repository ausliest und die Verzeichnisstruktur im Arbeitsverzeichnis nachbildet,

legt Subversion unter `.svn` noch eine Kopie des Projektes an. Dies geschieht, um zum Beispiel lokale Diffs, eine Anzeige der Unterschiede zweier Dateiversionen, durchführen zu können, ohne eine Verbindung zum Repository zu haben. Durch dieses doppelte lokale Abspeichern benötigt Subversion etwa doppelt so vielen lokalen Speicherplatz für ein Projekt, wie CVS benötigen würde, verzichtet dafür aber auf Netzwerkzugriffe bei Diffs.

11.2.2 Einchecken

- svn commit [Optionen] [Pfad]
- cvs commit [Optionen] [Pfad]

Hat man die Dateien verändert und möchte, dass die Änderungen auch den anderen zur Verfügung stehen, müssen diese Änderungen auch wieder in das Repository eingecheckt werden. Dies geschieht mit dem Befehl `commit`.
Bei der Ausführung dieser Operation unterscheiden sich CVS und Subversion wieder deutlich. Subversion benutzt das Konzept der „atomaren commits“, was bedeutet, dass es einen `commit`-Vorgang als eine Transaktion sieht und diesen nur durchführt, wenn alles übertragen werden kann. Bricht die Übertragung nach der vierten Datei ab, setzt Subversion das ganze Repository auf den Stand vor dem Übertragen zurück.
CVS hätte die ersten vier Dateien eingecheckt und dann abgebrochen, was dazu führen kann, dass unvollständige Änderungen im Repository liegen.
Bei Subversion dagegen liegt das Problem darin, dass, wenn der Abbruch unbemerkt bleibt, davon ausgegangen wird, alle Daten seien übertragen, dem aber nicht so ist. Wird nun das Arbeitsverzeichnis gelöscht, sind potenziell sämtliche Änderungen der Dateien irreparabel verschwunden. Aus diesem Grund sollte immer nachgesehen werden, ob die Änderungen im Repository gespeichert sind.

11.2.3 Updaten

- svn update [Optionen] [Pfad]
- cvs update [Optionen] [Pfad]

Der Befehl `update` ist dazu da, sein Arbeitsverzeichnis auf den neuesten Stand zu bringen. Ähnlich wie `checkout` holt er die Dateien aus dem Repository und überschreibt die alten Versionen mit den neuen. Kommt es zu Konflikten, werden Dateien „gemerged“, wie später in diesem Abschnitt beschrieben wird. Dateien und Verzeichnisse, die neu im Projekt sind, werden auf diese Weise auch ins Arbeitsverzeichnis geholt, um dann bearbeitet werden zu können.
Es ist auch möglich, ältere Versionen des Projektes zu bekommen, indem man dies bei den Optionen angibt.

11.2.4 Mergen

Bei beiden Programmen werden Dateien, die gleichzeitig verändert wurden, bei einem Update „gemerged“. Dies kann geschehen, wenn man comitten möchte, aber eine „veraltete Version“ der Datei hatte und diese geändert hat. Diese wird dann nicht comittet, und es ist nötig, ein Update zu ziehen. Wenn nur unterschiedliche Dinge geändert oder eingefügt wurden, wird einfach alles in die Datei geschrieben und man kann die Datei comitten. Wurden gleiche Zeilen geändert, werden beide Versionen in die Datei eingetragen und man muss diesen Konflikt „per Hand“ lösen. Am besten einigt man sich mit der Person, die die andere Änderung vorgenommen hat, auf eine Vorgehensweise, ändert die Datei entsprechend und checkt diese dann ins Repository ein.
Hier unterscheiden sich CVS und Subversion wieder. Während man bei CVS auch die unveränderte Datei mit beiden Versionen in einer Datei comitten kann, was auch aus Versehen passieren könnte, wartet Subversion auf ein `svn resolved`, welches die Lösung des Konfliktes signalisiert. Somit wird ein unerwünschter commit verhindert.
Nicht zu Verwechseln mit diesem Vorgang ist der Befehl `svn merge`. Dieser führt zwar auch wie oben beschrieben zusammen, allerdings geschieht dieses Mergen zwischen Branches, welche unterschiedliche „Abzweigungen“ in einem Projekt sind.

11.3 Komplexere Operationen

Dieser Abschnitt geht darauf ein, wie CVS und Subversion mit Dateien und Verzeichnissen umgehen, und wie in beiden eine Ignore-Option realisiert ist, bzw. was diese macht.

11.3.1 Umgang mit Dateien

Was das Umgehen mit Dateien angeht, sind die beiden betrachteten Programme unterschiedlich. Während CVS nur Unterschiede von Textdateien festhält, erstellt Subversion mit einem speziellem Algorithmus, auf den im Rahmen dieser Arbeit nicht näher eingegangen werden kann, Diffs auch auf Binärdateien. Subversion geht bei jeder Datei ersteinmal davon aus, dass es sich um eine Binärdatei handelt und erstellt das Diff und speichert dieses ab. Im Arbeitsverzeichnis wird zusätzlich gespeichert, ob es sich bei der Datei um eine Binärdatei oder eine Textdatei handelt. Bei einem Update werden nur die Änderungen an den Textdateien übertragen.
Mit Subversion ist es auch möglich, Dateien zu verschieben, zu löschen, zu kopieren und umzubenennen, ohne die „History“ zu verlieren, welche bei den Operationen, die CVS anbietet, verloren geht, falls die Operation überhaupt unterstützt wird.

11.3.2 Umgang mit Verzeichnissen

Der Umgang mit Verzeichnissen ist mit dem der Dateien vergleichbar. Unter Subversion ist es möglich, Verzeichnisse zu verschieben, zu löschen, zu kopieren und umzubennen und das Repository merkt sich den Ursprung. CVS lässt einmal vorhandene Verzeichnisse nicht mehr ändern.

11.3.3 Ignore-Optionen und Properties

CVS hat eine Datei `.cvsignore`. In diese kann man Dateinamen oder Dateiendungen, also zum Beispiel „readme*“ oder „*log“, eintragen, welche dann nicht mit eingecheckt werden, außer man erzwingt dieses.
Bei Subversion wird das Ganze mit einem *Property*, `svn:ignore`, gelöst. *Properties* sind Metadaten zu den Daten, die man versioniert, und werden ebenfalls mit versioniert. Im Falle des *Ignore-Properties* umgeht man das Fehlen einer `.svnignore` Datei. Mit `svn propset svn:ignore *log ./` erreicht man, dass für das aktuelle Verzeichnis alle Dateien, die auf `log` enden, nicht mehr comittet werden, außer man erzwingt dieses. Möchte man das Ganze als Datei bearbeiten, kann man mit dem Befehl `svn propedit` das ganze in einem Editor bearbeiten.
Weitere *Properties* sind zum Beispiel `svn:executable`, `svn:mime-type` und `svn:keywords` und noch einige mehr.

11.4 GUIs

GUIs gibt es als Standalone, integriert ins System, als Webinterface oder auch als Plug-in in einer Entwicklungsumgebung. Da der Rahmen dieser Arbeit doch sehr knapp bemessen ist, wird dieser Teil nur sehr kurz behandelt.

Ein Beispiel für ein integriertes System ist *Tortoise*. Dies ist ein Programm für Windows, das sämtliche Funktionen die man, je nach Installation mit CVS oder Subversion, nutzen kann, in den Explorer einbaut. Mit „cervisia“ gibt es etwas ähnliches für KDE. Es gibt noch weitere Programme für Linux und Windows, wie zum Beispiel rapidsvn, JSubversion und CvsGui.
Für die Entwicklungsumgebung Eclipse gibt es für beide Programme jeweils ein Plugin, was jedoch in einer anderen Ausarbeitung behandelt wird.
Ein sehr nützliches Programm, das man trotz des Namens für beide Systeme benutzen kann, ist CVSview. Es gibt auch Subversionview, welches man nur für SVN benutzen kann. Diese Werkzeuge bedienen sich eines Apache, um in der Lage zu sein, das Repository im Internet ansehbar zu machen. Dies ist nur ein lesender Zugriff, der aber in der Lage ist, Diffs und die Modulstruktur anzuzeigen.

11.5 Administration

Abschließend werden in diesem Abschnitt noch ein paar Punkte zur Administration behandelt, und was nötig ist, um mit CVS oder Subversion arbeiten zu können.

11.5.1 Repositories

Mit den Befehlen `cvs init` bzw. `svnadmin create` legt man ein Repository, mit `add` oder `import` dann die Modulstruktur an. Ein Repository kann mehrere Projekte enthalten, die unabhängig voneinander verwaltet und benutzt werden können.

11.5.2 „Modul-Aliase“

CVS kann sogenannte „Modul-Aliase“ verwalten. Das heißt, man kann einen Begriff definieren, der auf mehrere Verzeichnisse oder Dateien verweist, die man dann, unter Zuhilfenahme dieses Begriffs, gemeinsam auschecken kann. So eine Funktion ist in Subversion nicht implementiert, sondern muss durch Kopieren in ein anderes Verzeichnis gelöst werden. Auch wenn diese Kopie quasi nur ein Abbild ist, und keine vollständige Kopie, ist diese Umsetzung für Personen, die dieses CVS-Feature der Aliase zu schätzen wissen, eher unbefriedigend.

11.5.3 Zugriffskontrolle

In der Grundversion ist es bei beiden Programmen erst einmal nur möglich globale Rechte zu verteilen. Das heißt, jemand hat entweder gar keine Rechte, nur Leserechte oder Lese- und Schreibrechte für ein Repository.
Mit ausgeklügelten Mechanismen ist zumindest unter Subversion eine feinere Rechtevergabe möglich, da dies aber nicht zum Standard gehört, wird hier nicht näher darauf eingegangen.

11.5.4 Systemvorraussetzungen

Verzichtet man bei CVS auf die Webansicht, ist für die Benutzung nichts weiter nötig als das Programm selbst, wobei der Client beliebig ist. Man benötigt also nur ein wenig Speicherplatz. Möchte man zum Beispiel die Webansicht von CVSview nutzen, benötigt man auf Serverseite einen Apache.
Subversion ist etwas anspruchsvoller. Diese Versionsverwaltung benötigt, wie schon erwähnt, mehr Festplattenplatz und dazu noch, auf Serverseite, eine BerkeleyDB oder ein „FSFS“, welches nicht direkt eine Datenbank ist, aber die gleiche Funktion erfüllt. Dies weiter zu erklären, liegt nicht im Rahmen dieser Arbeit. Zu sagen ist dazu nur, dass die BerkeleyDB langsamer ist, mehr Speicherplatz benötigt und Probleme mit Netzwerk-Dateisystemen hat und es sich deswegen meist anbietet „FSFS“ zu benutzen. Für eine Webansicht benötigt auch Subversion auf der Serverseite einen Apache.

11.6 Fazit/Vergleich

Einer der Vorteile, die CVS ganz klar genießt, ist die Erprobtheit. Ein Programm, dass seid über 20 Jahren benutzt wird, hat einfach weniger Fehler. Dazu kommt, dass es keine Datenbank oder etwas Vergleichbares braucht und weniger Plattenplatz auf Clientseite benötigt.
Subversion dagegen ist zwar jünger und noch nicht so ausgereift, hat aber das Potential, weiterentwickelt zu werden, welches CVS einfach fehlt. Durch die Neuprogrammierung, anstatt des Aufsetzens auf RCS, ist es besser erweiterbar. Durch die doppelte lokale Kopie, die zugegebenermaßen mehr Speicherplatz belegt, ist es möglich, ohne ständigen Internetzugang gut zu arbeiten. Die „atomaren Commits“ bergen zwar ein paar Probleme, sind aber insgesamt gesehen vorteilhaft, da sie Inkonsistenzen vermeiden.

Desweiteren ist das Umgehen mit Binärdateien weitaus komfortabler. Durch die Operationen wie Kopieren, Verschieben usw. ist das Ganze leichter wartbar.

Teil II

Entwurfsdokumente

Dieser Teil des Endberichts enthält die Entwurfsdokumente von *Syspect*.

Die Produktskizze und die Anforderungsdefinition sind zwei Dokumente, die die funktionalen und nichtfunktionalen Anforderungen an das Produkt enthalten, ohne Lösungsvorschläge für die technische Umsetzung zu machen. Sie unterscheiden sich insofern, als dass die Produktskizze grob das Wunschprodukt der Kunden beschreibt, ohne dass hier aus Entwicklersicht technische Grenzen gesetzt werden. Nach der Reflexion über diesen ersten Schritt – sowohl von den Kunden als auch den Entwicklern – ist die Anforderungsdefinition das Ergebnis der Verhandlungen beider Seiten, bei denen es einerseits um möglichst konkret formulierte Forderungen und möglichst präzise abgeschätzte technische Machbarkeit geht.

Die hier getroffenen Vereinbarungen sind aber nicht absolut verbindlich, da sich Anforderungen während des gesamten Entwicklungsprozesses ändern können und auch die technische Umsetzung mancher Funktionalität schwieriger sein kann, als ursprünglich angenommen. Aus diesem Grund beinhaltet das nächste Kapitel die Änderungen, die im Laufe des Projektes in Bezug auf Produktskizze und Anforderungsdefinition vorgenommen wurden.

Der Entwurf schließlich beschreibt die Umsetzung von *Syspect*. Er enthält – aufgeteilt in die zwei Unterkapitel *Architektur* und *Feinentwurf* – die Dokumentation der wichtigsten Schritte, die während der Erstellung von *Syspect* durchgeführt wurden.

Kapitel 12

Produktskizze

12.1 Einleitung

Das vorliegende Kapitel soll als erste Beschreibung des zu entwickelnden Softwarewerkzeuges *Syspect* dienen, um die Kommunikation zwischen Entwicklern und Kunden zu erleichtern. Es hat nicht den verpflichtenden Charakter der auf diesem Kapitel aufbauenden Anforderungsdefinition und geht insbesondere nicht auf die technische Umsetzung ein.
Das Produkt der gleichnamigen Projektgruppe soll eine grafische Entwicklungsumgebung für ein UML-Modell sein, das durch ein UML-Profil derart eingeschränkt ist, dass eine Übersetzung in die formale Spezifikationssprache CSP-OZ-DC möglich ist. Die dazu nötigen Klassendiagramme, Strukturdiagramme und Zustandsdiagramme sollen über eine grafische Oberfläche bearbeitet werden können. Klassendiagramme geben Klassen und ihre Beziehungen zueinander wieder. In Klassendiagrammen können Operationen und Attribute der Klassen angegeben werden. Die Klassen besitzen sogenannte *tagged values*, die für die Übersetzung des UML-Modells in eine CSP-OZ-DC-Spezifikation nötigen Object-Z- und Duration Calculus-Formeln enthalten. Zustandsdiagramme beschreiben das Verhalten einer einzelnen Klasse und sollen in CSP-Prozesse umgewandelt werden. Die Strukturdiagramme stellen die Beziehungen der einzelnen Instanzen der Klassen dar [MORW05]. Aus dieser UML-Darstellung wird es möglich sein, eine XML-Darstellung der Spezifikationssprache CSP-OZ-DC zu erzeugen. Daraus kann anschließend LaTeX-Quellcode generiert werden. Zusätzlich soll das Werkzeug die Möglichkeit bieten, die CSP-OZ-DC-Spezifikation in Phasen-Event-Automaten zu transformieren, um deren Eigenschaften durch einen Model-Checker verifizieren zu können.

12.2 Die grafische Oberfläche der Entwicklungsumgebung

Die Entwicklungsumgebung ist in drei Bereiche aufgeteilt. Ein Vorschlag für das spätere Aussehen zeigt Abbildung 12.1.
Der Diagrammstrukturbereich beinhaltet eine baumartige Ansicht der verschiedenen

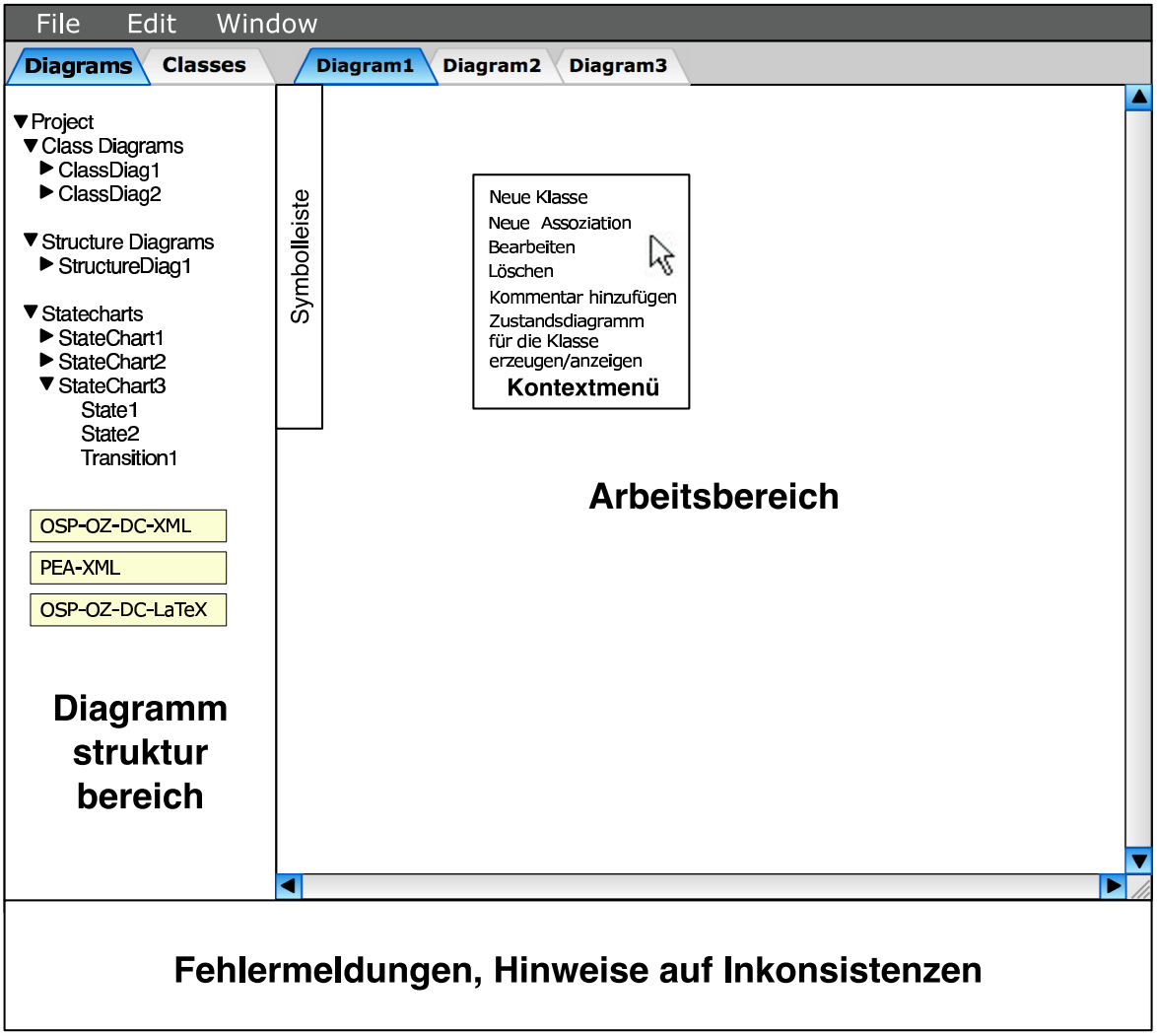


Abbildung 12.1: Schematische Übersicht der Entwicklungsumgebung

Diagrammarten und der einzelnen Komponenten der Diagramme. Neue Klassendiagramme können hier erstellt werden. Diese Funktion kann, wie auch alle weiteren, sowohl über das Kontextmenü als auch über das Programmmenü am oberen Bereich des Bildschirms erreicht werden. Weiterhin ist es möglich, sich alle Klassen der bis dahin erzeugten UML-Diagramme anzeigen zu lassen. Diese können innerhalb der Baumstruktur bearbeitet und per *Drag and Drop* in den Arbeitsbereich eingefügt werden. Am Ende des Diagrammstrukturbereichs befinden sich drei spezielle Buttons: *CSP-OZ-DC-XML*, *PEA-XML* und *CSP-OZ-DC-LaTeX*. Wird einer dieser Buttons geklickt, so wird die entsprechende Exportfunktion aufgerufen, so dass der Nutzer immer die aktuelle Übersetzung seines UML-Modells erhält.
Innerhalb des Arbeitsbereichs wird zu einem Zeitpunkt maximal ein Diagramm angezeigt. In diesem Bereich können die Elemente des jeweiligen Diagramms angeordnet, bearbeitet und erstellt werden. Es existiert eine Symbolleiste mit häufig benutzten Aktionen wie z.B. *Löschen*, *Neue Klasse*, *Neue Transition* usw. Desweiteren ist es mög-

lich, über ein Symbol innerhalb dieser Leiste den Vergrößerungsfaktor der Ansicht des Arbeitsbereiches zu verändern. Der untere Bereich der Oberfläche repräsentiert einen Nachrichtenbereich, in welchem Fehlermeldungen und Warnungen vor Inkonsistenzen zwischen den UML-Diagrammen ausgegeben werden.

12.2.1 Arbeiten mit Klassendiagrammen

Wurde ein neues Klassendiagramm angelegt, so beinhaltet dieses anfänglich keine Klassen. Sobald es geöffnet wurde, erscheint die Darstellung des Diagramms im Arbeitsbereich. Nun können neue Klassen und Beziehungen über die Symbolleiste, per *Drag and Drop* aus der Diagrammstruktur oder per Kontextmenü hinzugefügt werden. Innerhalb der Symbolleiste sind folgende Aktionen verfügbar:

- Neue Klasse
  - `capsule`
  - `protocol`
  - `data`
- Neue Assoziation
  - `base`-Aggregation
  - `conjugated`-Aggregation
  - Generalisierung
  - Komposition
- Bearbeiten
- Löschen
- Kommentar hinzufügen
- Zustandsdiagramm für die Klasse erzeugen/anzeigen

Beim Erzeugen einer neuen Klasse muss deren Stereotyp ausgewählt werden. Eine neue Klasse des ausgewählten Stereotyps erscheint daraufhin im Arbeitsbereich und kann hier angeordnet werden. Will der Benutzer eine Assoziation erstellen, so muss deren Typ ausgewählt werden, woraufhin die beiden zu verbindenden Klassen gewählt werden müssen. Innerhalb des Programms wird daraufhin eine Prüfung auf Konsistenz durchgeführt, deren Ergebnis im Nachrichtenbereich angezeigt wird. Die Wahl des Symbols *Bearbeiten* öffnet einen Dialog zum ausgewählten Element. In diesem können die relevanten Eigenschaften des Elements verändert werden, z.B. bei einer Klasse deren Name, Methoden, Attribute, Tags, usw. Zur Bearbeitung der Tags dient neben einer Texteingabe ein Formeleditor, der je nach Tag OZ- oder DC-Formeln erstellen kann. Optional kann sich der Bearbeitungs-Dialog in den Arbeitsbereich einbetten. Soll ein Element aus einem Diagramm gelöscht werden, so wählt man *Löschen*. Zum Hinzufügen eines Kommentars wird ein neues Kommentarelement erzeugt, in dem der

entsprechende Kommentartext eingegeben werden kann. Als zusätzliche Funktion ist es möglich, innerhalb der Klassendiagrammansicht ein Zustandsdiagramm für eine Klasse des Stereotyps `capsule` zu erzeugen. Durch erneutes Auswählen der Funktion oder bei schon existierendem Zustandsdiagramm, wird dieses in den Arbeitsbereich geladen und angezeigt.

12.2.2 Arbeiten mit Strukturdiagrammen

Das Verhalten des Programms beim Erzeugen eines Strukturdiagramms ist analog zum Erzeugen eines Klassendiagramms. Die folgenden Funktionen sind über die Symbolleiste erreichbar:

- Neue `capsule`
- Neuer `port`
  - `base`-Port
  - `conjugated`-Port
- `ports` verbinden
- Bearbeiten
- Löschen
- Kommentar hinzufügen
- Strukturdiagramm für die `capsule` erzeugen

Beim Erzeugen einer neuen `capsule` als Instanz einer Klasse des Stereotyps `capsule` wird ein Dialog geöffnet, in welchem die zu instanziierende Klasse ausgewählt werden kann. Eine `capsule` erscheint darauf im Arbeitsbereich. Wurden im Klassendiagramm `ports` spezifiziert (`base`-, `conjugated`-Aggregationen), so sind diese auch bei der erzeugten `capsule` vorhanden und können beliebig auf dem Rand dieser angeordnet werden. Durch Erzeugen eines neuen `ports` wird wiederum ein Dialog geöffnet, in welchem das zu verwendene `protocol` ausgewählt werden kann. Das Erstellen eines neuen `protocols` ist auch innerhalb des Dialogs möglich, woraufhin die entsprechene Klasse mit dem Stereotyp `protocol` erzeugt wird. Die Wahl des Symbols *Ports verbinden* erfordert das Auswählen zweier `ports`. Sind diese nicht über ein `protocol` in Beziehung so wird eine Warnung im dafür vorgesehenen Bereich der Oberfläche ausgegeben. *Bearbeiten*, *Löschen* und *Kommentar hinzufügen* verlaufen analog zur Beschreibung in Abschnitt 12.2.1. Als zusätzliche Funktion ist es möglich, innerhalb der Strukturdiagrammansicht ein Strukturdiagramm für das Innere einer `capsule` zu erzeugen. Falls die `capsule` schon ein Strukturdiagramm besitzt, wird dieses in den Arbeitsbereich geladen.

12.2.3 Arbeiten mit Zustandsdiagrammen

Auch beim Erzeugen eines Zustandsdiagramms ist das Verhalten des Programms analog zum Erzeugen eines Klassendiagramms. In der Symbolleiste sind die folgenden Funktionen verfügbar:

- Neuer Zustand
  - Initital-Zustand
  - Normaler Zustand
  - Endzustand
- Neue Transition
- Superzustand erzeugen
- Parallelität erzeugen
- Bearbeiten
- Löschen
- Kommentar hinzufügen

Wird ein neuer Zustand erzeugt, so muss zwischen einem Initialzustand, einem normalen Zustand und einem Endzustand unterschieden werden. Das Erstellen einer neuen Transition erfordert die Auswahl zweier zu verbindender Zustände und optional die zur Transition zugehörige Methode. Durch Auswahl des Symbols *Superzustand erzeugen* werden selektierte Zustände von einem Superzustand eingefasst. Eine andere Möglichkeit, einen Superzustand zu erschaffen, besteht darin, Zustände per *Drag and Drop* in einen anderen hineinzubewegen. Um Parallelität innerhalb eines Zustandes zu erzeugen, muss dieser vorher ausgewählt werden. Die Aktionen *Bearbeiten*, *Löschen* und *Kommentar hinzufügen* verlaufen wie in Abschnitt 12.2.1 beschrieben.

12.3 Exportfunktionen der Entwicklungsumgebung

Die Abbildung 12.2 veranschaulicht die Exportfunktionalitäten des Programms. Innerhalb des Dateisystems wird die UML-Darstellung nach dem XMI-Standard gespeichert, wodurch die Weiterverarbeitung durch XMI-unterstützende Programme ermöglicht wird. Wird die Funktion *Export nach CSP-OZ-DC* über die Menüleiste ausgewählt, so wird aus der internen UML-Darstellung die CSP-OZ-DC-Spezifikation erstellt. Diese liegt anschließend in einem XML-Format vor. Außerdem ist es möglich, direkt LaTeX-Quellcode oder eine XML-Darstellung von Phasen-Event-Automaten (PEA) zu generieren. Diese Darstellung von PEA kann daraufhin von einem Model-Checker auf Korrektheit verifiziert werden.

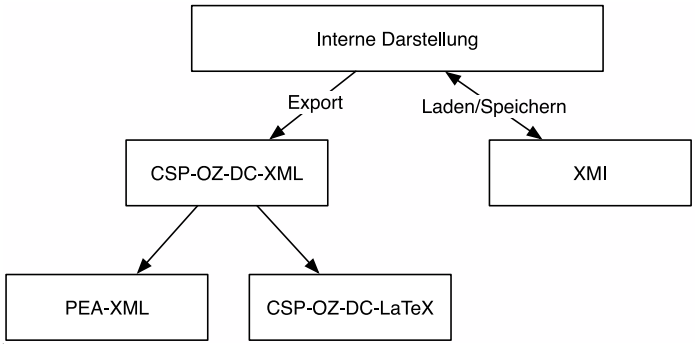


Abbildung 12.2: Schematische Übersicht über die Exportfunktionen

Kapitel 13

Anforderungsdefinition

13.1 Einleitung

Dieses Dokument beschreibt die Anforderungen an das Werkzeug Syspect, indem es die in der Produktskizze (Kapitel 12) enthaltenen Beschreibungen präzisiert. Das Produkt soll eine grafische Entwicklungsumgebung für ein UML-Modell sein, das durch ein UML-Profil derart eingeschränkt ist, dass eine Übersetzung in die formale Spezifikationssprache CSP-OZ-DC [HM05] möglich ist. Hierdurch wird dem UML-Modell auch eine eindeutige Semantik zugeordnet. Das UML-Profil wurde aus dem Seminar *Ein UML-Profil für CSP-OZ-DC* (siehe Kapitel 9, S. 77) übernommen. Aus der UML-Darstellung soll eine XML-Darstellung der Spezifikationssprache CSP-OZ-DC erzeugt werden. Daraus soll anschließend LaTeX-Quellcode generiert werden können. Zusätzlich wird das Werkzeug die Möglichkeit bieten, die CSP-OZ-DC-Spezifikation in Phasen-Event-Automaten (PEA) [HM05] zu transformieren. Die Eigenschaften dieser PEA sollen sich daraufhin durch einen externen Model-Checker verifizieren lassen.

13.2 GUI-Darstellung der UML-Diagramme

Für die Spezifikation des UML-Modells werden Klassendiagramme, Zustandsdiagramme und Strukturdiagramme verwendet. Den Diagrammen liegt eine interne Struktur zugrunde, die das gesamte spezifizierte Modell beinhaltet. Die Gestaltung der Diagramme ist einheitlich und durchgängig gehalten. Alle Funktionen, die auf Diagramme angewendet werden können, verfügen generell über eine *History*-Funktion. Ebenso bedarf es für die Diagramme einer Druckansicht und einer Druckfunktion. Des Weiteren besteht die Möglichkeit, die Diagramme in die Formate *Portable Network Graphics* (PNG) und *Encapsulated Postscript* (EPS) zu exportieren. Zur Unterstützung des Entwicklungsprozesses gibt es eine Hilfefunktion sowie eine Verbindung von Metadaten zu Diagrammen. Eine weitere Unterstützung wird durch aussagekräftige Fehlermeldungen gegeben. Diese Fehlermeldungen können aufgrund von fehlerhaften Benutzereingaben oder Inkonsistenzen in verschiedenen, zusammengehörenden Diagrammen auftreten.

13.2.1 Metadaten

1. Um den Entwicklungsprozess und die Dokumentation zu unterstützen, ist es möglich, sowohl für die Dokumentation als auch für den Entwicklungsprozess Informationen zu erstellen. Die Informationen werden allgemein als Metadaten bezeichnet. Metadaten enthalten Kommentare, Beschreibungen sowie Arbeitsaufträge. Sie können mit anderen Elementen, bspw. einer Klasse, verbunden werden.

2. Metadaten können erstellt, verändert und gelöscht werden. Die Bearbeitung ist an zwei Stellen möglich:

   (a) innerhalb der grafischen Darstellung des Teilbereichs, bspw. einer Klasse,

   (b) in einem Eingabebereich außerhalb der grafischen Darstellung des Teilbereichs.

13.2.2 Klassendiagramme

Es besteht die Möglichkeit, Klassendiagramme zu erstellen, zu bearbeiten und anzuzeigen. Ein Klassendiagramm stellt einen Teil der Sprache CSP-OZ-DC dar. Für die grafische Darstellung wird das aus der UML stammende Klassendiagramm [UML05] an CSP-OZ-DC entsprechend dem UML-Profil angepasst. Ein Klassendiagramm kann Klassen, Assoziationen und Metadaten enthalten.

Klassendiagramm

1. Klassendiagramme werden in einem festgelegten Anzeigebereich des Syspect-Werkzeugs dargestellt. Zwischen verschiedenen geöffneten Klassendiagrammen kann mit Hilfe von Reitern gewechselt werden.

2. Ein Klassendiagramm besitzt einen Namen und kann Metadaten integrieren. Die Bearbeitung ist an zwei Stellen möglich:

   (a) innerhalb der grafischen Darstellung der Klasse selbst,

   (b) in einem Eingabebereich außerhalb der grafischen Darstellung.

   Ein Klassendiagramm kann gelöscht werden.

3. Die Darstellung der Klassendiagramme ist mit den Strukturdiagrammen konsistent. Diesbezüglich führen Änderungen in Strukturdiagrammen automatisch zu Änderungen in Klassendiagrammen und Änderungen in Klassendiagrammen automatisch zu Änderungen in Strukturdiagrammen.

4. Ein Klassendiagramm gibt lediglich eine eingeschränkte Sicht auf das zugrunde liegende UML-Modell wieder. Das Löschen von Klassen oder Assoziationen führt nicht zwangsläufig zur Veränderung des zugrunde liegenden UML-Modells, wenn andere Klassendiagramme diese Klassen und Assoziationen immer noch anzeigen. Das Verändern des UML-Modells geschieht erst, nachdem in sämtlichen Sichten auf das UML-Modell die Veränderung durchgeführt wurde. Optional wird es die

Möglichkeit geben, eine Klasse oder Assoziation in allen Klassendiagrammen zu löschen, so dass auch das entsprechende Element im UML-Modell gelöscht wird.

5. Zur Vergrößerung bzw. Verkleinerung der Darstellung eines Klassendiagramms gibt es eine Zoom-Funktion.

Klasse

1. Klassen werden innerhalb eines Klassendiagramms grafisch dargestellt. Die Darstellung einer Klasse kann in ihrer Größe verändert werden.

2. Eine Klasse besitzt einen Namen und verfügt über einen Stereotyp sowie über Attribute, Tags und Methoden. Der Stereotyp einer Klasse kann `capsule`, `protocol` und `data` sein. Attribute haben einen Namen und einen Datentyp. Methoden besitzen ebenfalls einen Namen und können über Parameter, Tags sowie Rückgabewerte verfügen. Zu dem Stereotyp `capsule` gehören die Tags `invariant`, `init` und `dc-counterexample`. Die Methoden einer `capsule` enthalten jeweils die Tags *enable*, *effect* und *changes*. Die Methoden einer Klasse des Stereotyps `protocol` besitzen die Tags *input*, *output* und *simple* und den Stereotyp `in`, `out`, `in_s` oder `out_s`. Eine Klasse integriert auch Metadaten.

3. Optional: Eine Klasse kann als Schablone gespeichert werden. In der Schablone werden alle Inhalte einer Klasse gespeichert. Solche Schablonen dienen zur Wiederverwendung von Klassen und unterstützen so die Erstellung von Klassendiagrammen. Falls diese Anforderung umgesetzt werden soll, ist noch genauer zu definieren, inwiefern sich Schablonen von „normalen“ gespeicherten Klassen unterscheiden.

4. Klassen können erstellt und gelöscht werden. Bei der Erstellung wird automatisch eine leere Klasse innerhalb eines Klassendiagramms grafisch dargestellt.

5. Die Bearbeitung einer Klasse ist an zwei Stellen möglich:

   (a) innerhalb der grafischen Darstellung der Klasse selbst

   (b) und in einem Dialogfenster.

   Die Bearbeitung wird dabei insofern unterstützt, dass vorhandene Methoden, Attribute und Tags zur Auswahl stehen bzw. die Auswahl auch auf den Stereotyp einer Klasse angepasst wird. Für die Bearbeitung einer Methode einer Klasse ist eine spezielle Darstellung gegeben (siehe Abschnitt 13.2.2).

6. Weiter besteht die Möglichkeit, dass zu einer Klasse ein Zustandsdiagramm erstellt werden kann. Existiert ein solches schon, kann dieses in den Arbeitsbereich geladen werden.

7. Die Darstellung ist mit dem Zustandsdiagramm konsistent. Diesbezüglich führen Änderungen im Zustandsdiagramm automatisch zu Änderungen im Klassendiagramm und Änderungen im Klassendiagramm automatisch zu Änderungen im

Zustandsdiagramm. Es ist jedoch zu beachten, dass in verschiedenen Diagrammen ein und dasselbe Element unterschiedliche Eigenschaften haben kann, da jedes Diagramm nur eine eingeschränkte Sicht auf das zugrunde liegende UML-Modell bietet.

Assoziationen

1. Assoziationen werden innerhalb eines Klassendiagramms grafisch dargestellt. Über eine Assoziation werden genau zwei Klassen miteinander verbunden. Assoziationen können frei oder automatisch ausgerichtet werden.

2. Eine Assoziation verfügt über eine Beschriftung, Stereotypen und Kardinalitäten. Ein Stereotyp kann `composition`, `aggregation` oder `generalization` sein. Zudem kann eine Assoziationen als `base` (Sender) oder `conjugate` (Empfänger) dienen. Eine `aggregation` ist nur zwischen zwei Klassen der Stereotypen `capsule` und `protocol` erlaubt. Eine `composition` ist zwischen zwei Klassen der Stereotypen `capsule` und `data` oder `capsule` und `capsule` möglich.

3. Eine Assoziation kann erstellt, bearbeitet und gelöscht werden. Die Bearbeitung soll an zwei Stellen möglich sein:

   (a) innerhalb der grafischen Darstellung der Assoziation selbst und

   (b) in einem Dialogfenster.

Object-Z-Datentypen und -Ausdrücke und DC-Ausdrücke

1. Es gibt sowohl einen Editor für die Eingabe von Object-Z-Datentypen und -Ausdrücken als auch einen für die Eingabe von DC-Ausdrücken. Die Eingabe wird durch die Unterstützung von Fehlermarkierungen und gegebenenfalls Syntax-Highlighting sowie der Erzeugung von ausgewählten Ausdrücken per GUI in den beiden Editoren erleichtert. Auf die Eingabe wird ebenfalls eine Typprüfung ausgeführt, die Typverletzungen aufzeigt.

2. Die Eingabe von Countertraces und Testformeln wird durch einen DC-Formeleditor erleichtert.

3. Der Editor unterstützt eine LaTeX-Eingabe für Object-Z- und DC-Ausdrücke. Optional ist eine Code Completion für diese Eingabe vorgesehen.

4. Zur besseren Lesbarkeit des eingegebenen Codes gibt es eine äquivalente Unicode-Darstellung zu diesem Code. Die Unicode-Darstellung zeigt somit bestimmte Eingabezeichen und LaTeX-Eingaben, die nicht direkt über die Tastatur eingegeben werden können, in ihrem Unicode-Zeichen wie bspw. $\geq$. Die Unicode-Darstellung wird jederzeit mit der Eingabe synchronisiert.

13.2.3 Zustandsdiagramme

Zu jeder vorhandenen `capsule` soll ein Zustandsdiagramm [UML05] erstellt werden können. Sie werden genutzt, um das Verhalten einer `capsule` zu modellieren. Neben dem Anlegen, Ändern und Löschen von Zustandsdiagrammen ist eine Eingabe von Kommentaren vorgesehen. Kommentare sind für die Semantik irrelevant, sie sollen lediglich den Entwickler bei seiner Arbeit unterstützen.

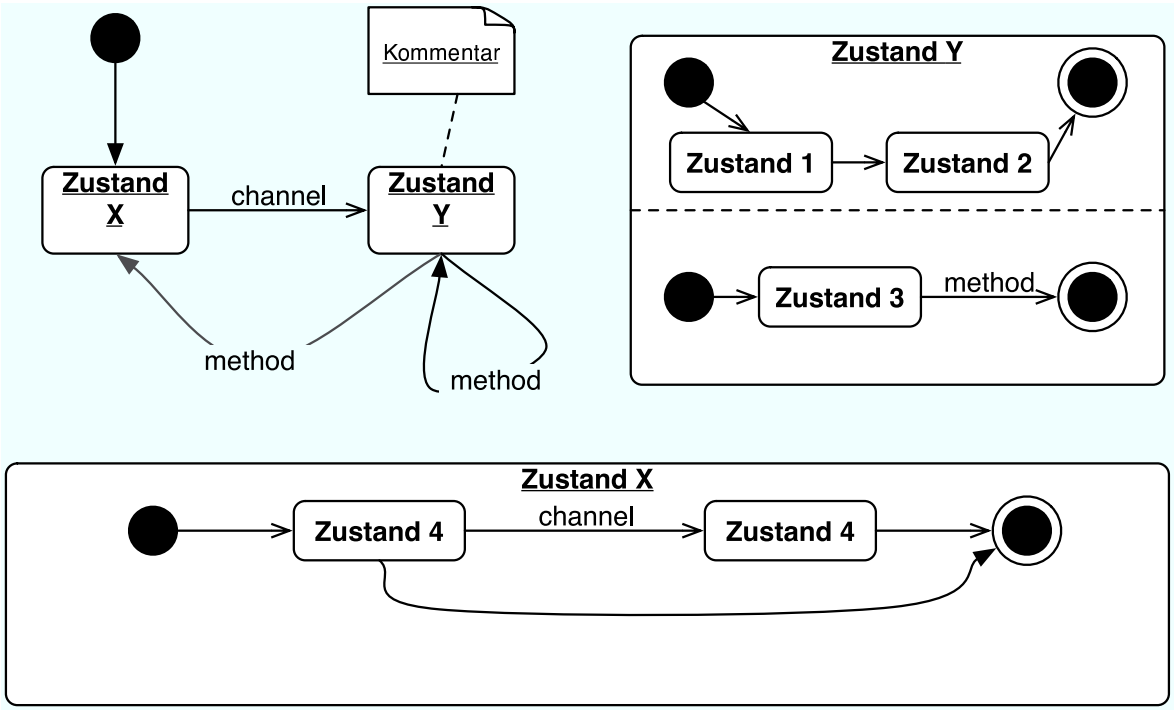


Abbildung 13.1: Elemente von Zustandsdiagrammen

Allgemeine Eigenschaften

1. Da jedes Zustandsdiagramm das Verhalten einer Klasse des Stereotyps `capsule` beschreibt, werden neue Zustandsdiagramme in der Klassendiagrammansicht angelegt. Soll ein Zustandsdiagramm angelegt werden, muss zunächst eine entsprechende Klasse erstellt werden.

2. Der Name eines Zustandsdiagramms leitet sich von dem Namen der zugehörigen Klasse ab.

3. Die Darstellung der Zustandsdiagramme wird jederzeit mit dem internen Modell konsistent gehalten.

4. Es besteht die Möglichkeit, die Ansicht zwischen einem Zustandsdiagramm und dem zugehörigen Klassendiagramm zu wechseln.

5. Ein Zustandsdiagramm kann aus den Diagrammelementen bestehen, die in Abbildung 13.1 zu sehen sind. Eine nähere Beschreibung der Semantik kann aus [MORW05] und [MORW04] entnommen werden.

6. Die verschiedenen Zustände (Start-, End-, Normal- und Superzustand) sowie Transitionen und Kommentare können dem Diagramm hinzugefügt bzw. wieder aus diesem gelöscht werden.

7. Zustände können einem Superzustand hinzugefügt bzw. wieder aus diesen herausgenommen werden.

8. Superzustände können unterteilt werden, so dass es möglich ist, parallele Abläufe im Sinne von CSP zu definieren. Diese Unterteilung kann hinzugefügt bzw. wieder gelöscht werden.

9. Die Entwicklungsumgebung gibt Hinweise auf mögliche Fehler. Hierzu zählen

   (a) ein fehlender Startzustand,

   (b) eine fehlende Transition vom Startzustand,

   (c) Zustände, auf die keine Transitionen zeigen und

   (d) Beschriftungen von Transitionen mit Operationen, die in den zugehörigen Klassen nicht definiert sind.

13.2.4 Strukturdiagramme

In Ergänzung zum erwähnten Klassendiagramm sollen die Strukturdiagramme unterstützt werden. Strukturdiagramme dienen in diesem Projekt zur Modellierung der Kommunikationsstruktur instanziierter `capsule`-Klassen. Neben dem Anlegen, Bearbeiten und Löschen soll diese Diagrammart ebenfalls die Eingabe von Metadaten unterstützen. Eine Übersicht über die grafischen Elemente dieser Diagrammart zeigt Abbildung 13.2.

Allgemeine Eigenschaften

1. Die Erstellung eines Strukturdiagramms kann aus der Klassendiagramm-Ansicht heraus erfolgen. Hierfür wird dort eine vorhandene Klasse zur Instanziierung ausgewählt. Das instanziierte Objekt stellt nun das äußere Objekt dar. Objekte, die durch eine Komposition an das instanziierte Objekt gebunden sind, werden im Inneren dieses Objekts angezeigt. Ebenfalls werden die Ports der Objekte angezeigt.

2. Der Name eines Strukturdiagramms wird von dem Namen der zugehörigen Klasse abgeleitet. Die Bearbeitung kann sowohl über die grafische Darstellung als auch über externe Eingabebereiche erfolgen. Zudem sollte eine Auswahlliste mit allen Strukturdiagrammen vorhanden sein.

3. Die Darstellung wird jederzeit mit dem Gesamtmodell konsistent gehalten.

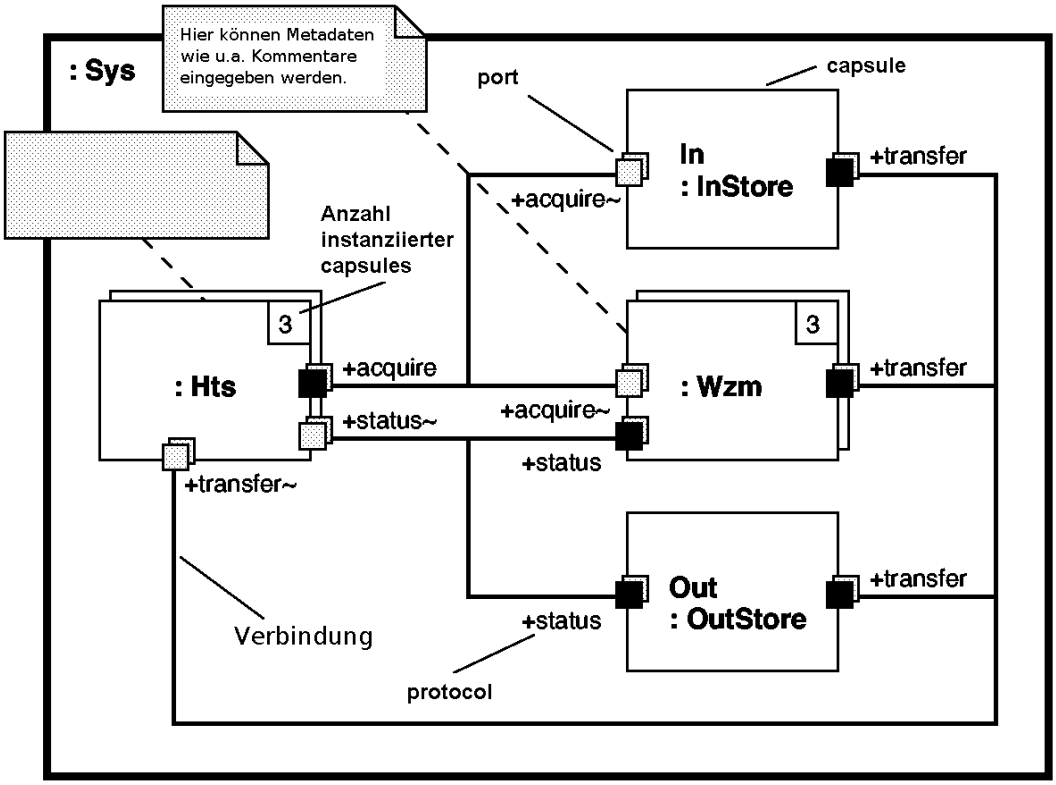


Abbildung 13.2: Diagrammelemente eines Strukturdiagramms

Eigenschaften von Objekten

1. Ein Objekt ist eine Instanz einer Klasse mit dem Stereotyp `capsule`.

2. Ein Objekt kann hinzugefügt, bearbeitet oder gelöscht werden.

Eigenschaften von Ports

1. Die Ports von Objekten entsprechen im Klassendiagramm einer Aggregationsbeziehung zwischen der zugehörigen Klasse und einer Klasse des Stereotyp `protocol`.

2. Ports sind Verbindungsstellen, mit denen eine Kommunikationsverbindung unter einzelnen Objekten möglich ist. Ports sind in der Ausprägung `base` und `conjugated` verfügbar und können den Objekten hinzugefügt oder aus diesen gelöscht werden. Der Benutzer muss einem neuen Port direkt eines der vorhandenen Protokolle zuordnen bzw. ein Protokoll neu erstellen.

3. Sollten Ports, an denen Verbindungen bestehen, gelöscht werden, so wird die entsprechende Verbindung auch gelöscht.

Verbindungen

1. Es können ausschließlich Ports miteinander verbunden werden, die dasselbe Protokoll sprechen.

2. Verbindungen zwischen einem äußeren und einem inneren Objekt können nur zwischen Ports des gleichen Typs bestehen (`conjugated`/`conjugated` bzw. `base`/`base`)

3. Eine Verbindung kann hinzugefügt, bearbeitet oder gelöscht werden.

13.3 Nichtfunktionale Anforderungen

13.3.1 Performance

In diesem Abschnitt spezifizieren wir Anforderungen an das zeitliche Verhalten des Werkzeugs. Die zeitlichen Angaben beziehen sich auf einen Athlon XP 1700+, 512MB RAM, SuSE Linux, Java 5.0. Das Werkzeug Syspect soll so entwickelt werden, dass die angegebenen Zeiten mit den *Java-Event-Handlern* eingehalten werden.

Das Laden von Dateien sollte innerhalb von maximal 3 Sekunden geschehen. Das Sichern einer durchschnittlichen Spezifikation von Größe der in [Fab05] beschriebenen Fallstudie sollte nicht mehr als 3 Sekunden beanspruchen. Falls es möglich ist, während des Vorgangs weiterzuarbeiten ohne Inkonsistenzen hervorzurufen, kann diese Zeit überschritten werden. Die angestrebte Umwandlung von UML in CSP-OZ-DC-XML und weiter in LaTeX sollte auch nicht mehr als 10 Sekunden beanspruchen. Die Reaktion der GUI dagegen sollte möglichst zeitnah zur Eingabe der Benutzers erfolgen. Das bedeutet, dass zwischen Nutzeraktion und Reaktion des Programms nicht mehr als 5 ms vergehen sollten. Die Diagrammaktualisierung bzw. die Syntax- und Semantik-Überprüfung sollte nach jeder abgeschlossenen Eingabe, also z.B. nach jedem Druck auf die Eingabetaste, erfolgen, um Fehler direkt sichtbar zu machen. Für Texteingaben gilt, dass jedes Zeichen eingeblendet werden sollte, bevor das nächste getippt wurde. Zur Überprüfung dieser Angaben wird von einer Tippgeschwindigkeit von 300 Zeichen pro Minute ausgegangen. Bei der Darstellung von Sonderzeichen, also beispielsweise der Ersetzung von Eingaben in entsprechende Z-Symbole, darf diese Zeit überschritten werden – und zwar so, dass minimal eineTippgeschwindigkeit von 100 Zeichen pro Minute möglich ist.
Der Start des Programms darf den Start des Basis-Programms nicht um mehr als 3 Sekunden verlängern. Das Werkzeug sollte nicht nur beim Speichern und Laden sondern auch bei kritischen Programmfehlern konsistente Zustände im Dateisystem hinterlassen. Das Programm darf darüber hinaus maximal einmal pro Arbeitstag (8 Stunden) abstürzen. Bei einem Absturz darf maximal die Arbeit der letzten 10 Minuten verloren gehen.

13.3.2 Sonstiges

1. Das Programm sollte unter die *General Public License* (GPL) oder eine freiere Lizenz gestellt werden.

2. Die Konventionen zur Quellcodeformatierung und -Kommentierung muss mit den Kunden abgesprochen sein, um einfache Erweiterungen zu ermöglichen.

13.4 Dateiformate

Dieser Abschnitt ergänzt und erweitert Kapitel 3 der Produktskizze (Kapitel 12) ("Exportfunktionen der Entwicklungsumgebung").

13.4.1 XML-Darstellung der CSP-OZ-DC-Spezifikationen

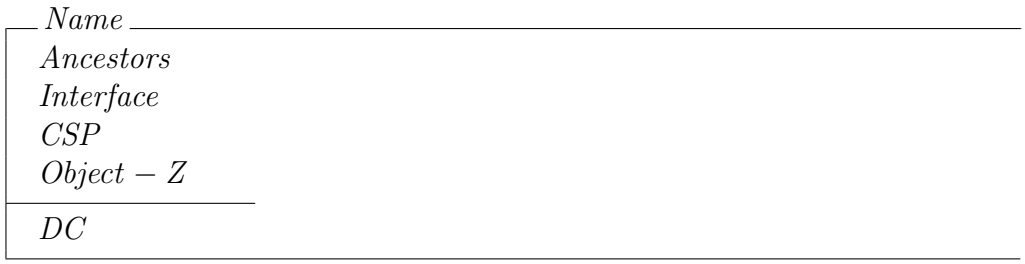


Abbildung 13.3: Aufbau einer CSP-OZ-DC-Klasse

Die UML-Darstellung einer CSP-OZ-DC-Spezifikation soll in eine XML-Repräsentation für CSP-OZ-DC exportiert werden können. Dieses XML-Format für CSP-OZ-DC wird weiter benötigt, um daraus eine XML-Darstellung von Phasen-Event-Automaten oder LaTeX-Quellcode für die Darstellung der Spezifikation zu generieren.

1. Abbildung 13.3 zeigt den Aufbau einer CSP-OZ-DC-Klasse, welche in die Abschnitte Ancestors, Interface, CSP, Object-Z und Duration Calculus aufgeteilt ist.Die Definition einer CSP-OZ-Klasse ist aus [MORW05] übernommen, wobei der Klasse noch ein Abschnitt für DC-Gegenbeispielformeln hinzugefügt wurde. Die einzelnen Abschnitte sollen strukturiert im CSP-OZ-DC-XML-Schema vorhanden sein. Auf der obersten Ebene des XML-Dokuments werden globale Definitionen und Klassen aufgeführt.

2. Beispielhaft ist der grobe Aufbau des XML-Dokumentes einer CSP-OZ-DC-Klasse `test` in Abbildung 13.4 dargestellt.

3. Die Darstellung der Object-Z-Ausdrücke soll in ZML erfolgen.

4. Die Darstellung der Duration Calculus Formeln soll möglichst mit der Darstellung von Roland Meyer [Mey05b] übereinstimmen.

5. Es soll möglich sein, Object-Z-Datentypen sowohl auf globaler Ebene als auch innerhalb der einzelnen Klassen zu definieren.

```
<?xml version="1.0" encoding="UTF-8"?> <cod
xmlns:xsi="...[Namespace einsetzen]...">
  <class name="test">
    <ancestors>
      <inherit>Junit</inherit>
    </ancestors>
    <interface>
      <method name="dummy">
        <param type="PAR">par?</param>
      </method>
      <channel name="output">
        <param type="PAR">out!</param>
        <param type="TestRef">adr</param>
      </channel>
    </interface>
    <local>
      <channel name="intern"/>
    </local>
    <csp>
      ...
    </csp>
    <oz>
      ...siehe ZML...
    </oz>
    <dc>
      ...siehe Darstellung von Roland Meyer
    </dc>
  </class>
</cod>
```

Abbildung 13.4: XML-Beispiel einer CSP-OZ-DC-Klasse

13.4.2 XML-Darstellung von Phasen-Event-Automaten

Im Folgenden werden die Anforderungen an ein XML-Format zur Darstellung von Phasen-Event-Automaten (PEA) definiert.

Anforderungen an das XML-Format

1. Es ist nötig, den Zustandsraum, Zustandsvariablen, Events, Uhren, Invariantenprädikate sowie Uhreninvarianten für Zustände in XML zu kodieren. Die Tran-

sitionsrelation inklusive Guards, Events und Uhrenrücksetzungen muss in XML ausgedrückt werden. Die Guards sind hierbei Z-Prädikate. Es muss möglich sein, Transitionen als Initialtransitionen zu definieren.

2. Die Formeln, die hierbei auftreten, beispielsweise in den Invarianten, sollen nicht nur als `String` gespeichert werden, sondern ihre Struktur soll in eine typische XML-Schreibweise überführt werden.

3. Object-Z-Datentypen sollen nicht nur als `String` verfügbar sein, sondern strukturiert in XML definiert werden können.

4. Die Ausdrücke in Object-Z sollen in diesem Format entweder identisch mit denen in der XML-Darstellung der CSP-OZ-DC-Spezifikation sein oder in einer Form dargestellt werden, die sich an dem Format orientiert, in das die PEA weiter konvertiert werden.

5. Des Weiteren soll es möglich sein, mehrere PEA zu einem Automatenverbund zusammenzufassen.

Beschreibung des XML-Formats für Phasen-Event-Automaten

Das aktuell bereits vorhandene Format [Mey05b] für die Darstellung von PEA soll genutzt werden und darf nur in Absprache mit dem Auftraggeber angepasst werden. Die Struktur der XML-Darstellung gliedert sich folgendermaßen:
Auf der obersten Ebene wird ein `PEANet` definiert. Dies ist ein Automatenverbund. Innerhalb des PEANets gibt es ein oder mehrere PEA. Ein PEA enthält eine Variablenliste, eine Eventliste, eine Uhrenliste, eine Zustandsliste und eine Transitionsliste. Transitionen enthalten auf jeden Fall eine Quelle und ein Ziel, außerdem können sie Guards, als `Formula`, und eine Menge von Uhren, die zurück auf null gesetzt werden sollen, enthalten. Zustände haben den Typ `PEAPhase` und setzen sich aus Variableninvarianten und Uhreninvarianten, beides als `Formula`, sowie einem Namen und einem `Boolean`, das angibt, ob sie Startzustand sind oder nicht, zusammen.
Hierbei werden folgende Basisdatentypen benutzt. Da die Basisdatentypen auch für andere Schemata benutzt werden, folgt hier nun die Auflistung der für das PEA-Format relevanten:

1. `Formula`: Eine `Formula` besteht entweder aus genau einer `BooleanExpression`, genau einer `RangeExpression`, genau einer `EventExpression` oder genau einem `FormulaTree`. Es können also boolsche Formeln damit kodiert werden.

2. `BooleanExpression`: Eine `BooleanExpression` dient zum Notieren boolscher Ausdrücke. Im Moment wird der Ausdruck jedoch nur als `String` kodiert.

3. `RangeExpression`: Eine `RangeExpression` dient zum Vergleich einer Variable mit einer Zahl (`double`).

4. `FormulaTree`: Ein `FormulaTree` besteht aus einer bis unbegrenzt vielen `BooleanExpression`, `RangeExpression` `EventExpression`, `FormulaTree` und einem `FormulaOperator`.

5. `RangeOperator`: Der `RangeOperator` ist eine binäre Relation ($>, \geq, =, \neq, \leq, <$). Im XML-Format werden diese Symbole durch `Strings` repräsentiert (*greater*, *greaterequal*, *equal*, *notequal*, *lessequal*, *less*).

6. `EventExpression`: Eine `EventExpression` speichert den Namen eines Events.

7. `FormulaOperator`: Der `FormulaOperator` kann im Moment AND, OR oder NOT sein. Er wird verwendet, um boolsche Formeln in XML kodieren zu können.

8. `Variable`: Eine `Variable` besteht aus einem Namen und einem Typen (beides als `String`).

9. `Clock`: Eine `Clock` besteht nur aus einem Namen als `String`.

Die Abbildung 13.5 enthält ein Beispiel für das aktuelle Format.

Ansatzpunkte für Änderungen

Da eine Anforderung die Erweiterung des aktuellen Formates ist, hier eine Liste von Ansatzpunkten für die notwendigen Erweiterungen.

1. Variablentypen müssen vom Benutzer definierbar sein. Im Moment ist die Typdarstellung nur als `String` vorhanden.

2. Mit dem `FormulaTree` ist im aktuellen Format nur die Darstellung von boolschen Ausdrücken und binären Relationen möglich. Hier soll die Darstellung von Z-Ausdrücken ermöglicht werden.

3. Es muss möglich sein, Transitionen mit nur einem Ziel, sowie optional Guards und der Uhrenrücksetzmenge einzufügen, da nach der neusten PEA-Definition [Hoe05] die initialen Kanten beschriftet werden dürfen.

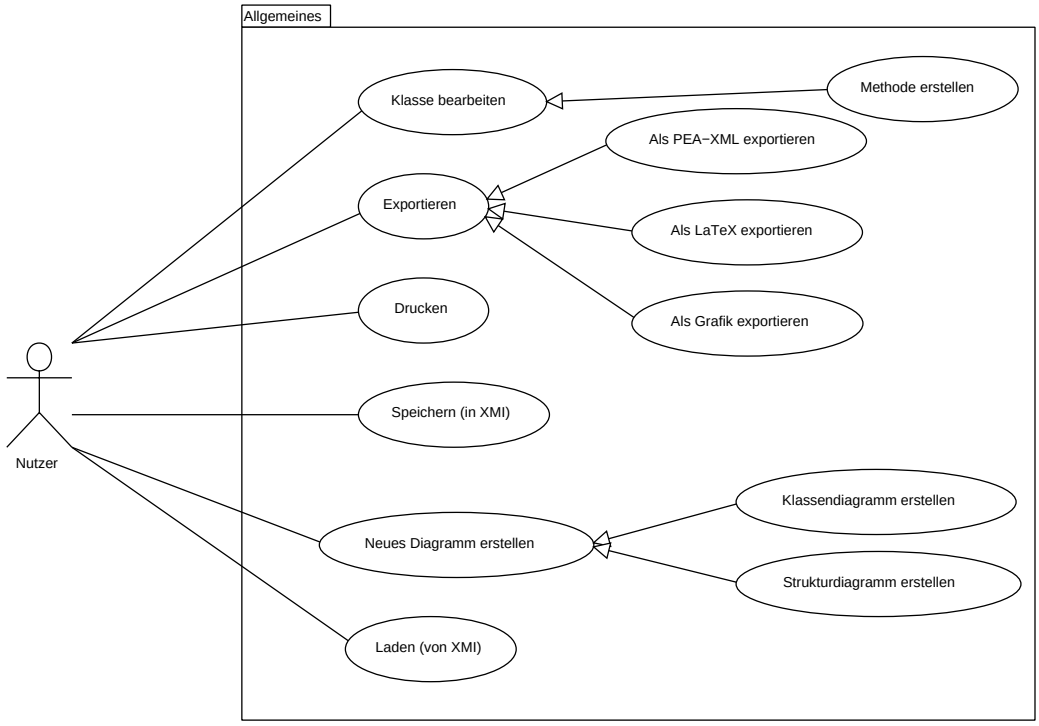
13.5 Anwendungsfälle

Um das Arbeiten mit dem Werkzeug Syspect zu verdeutlichen, sind analog zur Produktskizze zu vier Themenbereichen Anwendungsfalldiagramme mit Erläuterungen vorhanden. Diese decken jedoch nur oberflächlich den jeweiligen Anwendungsfall ab. Eine genauere Erläuterung inklusive von Konsistenzbedingungen werden im Entwurf erarbeitet.
Die Anwendungsfalldiagramme für das Bearbeiten von Klassen-, Struktur und Zustandsdiagrammen beschreiben die möglichen Anwendungsfälle im Umgang mit dem jeweiligen Diagrammtyp. Das allgemeine Anwendungsfalldiagramm befasst sich mit Drucken, Speichern oder Laden usw. – also Fällen, die nicht mit der unmittelbaren Bearbeitung eines Diagramms zu tun haben, sondern allgemeingültige wichtige Funktionen erfüllen.

```
<?xml version="1.0" encoding="UTF-8"?>
  <peaNet xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation = "../schemas/PEA.xsd">
    <pea name = "pea2">
      <variables>
        <variable name = "hund" type = "default_type"/>
        <variable name = "hun" type = "default_type"/>
      </variables>
      <events>
        <event name = "S0"/>
      </events>
      <phases>
        <phase initial = "true" name = "st">
          <invariant>
            <booleanExpression expression = "true"/>
          </invariant>
        </phase>
        <phase initial = "true" name = "st0">
          <invariant>
            <formulaTree operator = "AND">
              <rangeExpression bound = "5" operator = "lessequal"
                variable = "hund"/>
              <booleanExpression expression = "a"/>
            </formulaTree>
          </invariant>
        </phase>
      </phases>
      <transitions>
        <transition source = "st0" target = "st">
          <guard>
            <formulaTree operator = "AND">
              <formulaTree operator = "NOT">
                <eventExpression name = "S0"/>
              </formulaTree>
              <rangeExpression bound = "5"
                operator = "lessequal" variable = "hund"/>
            </formulaTree>
          </guard>
        </transition>
      </transtions>
    </pea>
  </peaNet>
```

Abbildung 13.5: Aktuelles XML-Format für Phaseneventautomaten

13.5.1 Allgemeine Anwendungsfälle



Anwendungsfall: LaTeX-Exportieren

Akteure:
Nutzer

Ziel:
LaTeX-Code generieren

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Exportieren nach LaTeX" ausgewählt.

Vorbedingung:
Es muss (automatisch oder manuell) die CSP-OZ-DC-XML-Datei generiert worden sein.

Nachbedingung Erfolg:
Es existiert der LaTeX-Code.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Die CSP-OZ-DC-XML-Datei.

Anwendungsfall: PEA-XML Exportieren

Akteure:
Nutzer

Ziel:
PEA-XML generieren

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Exportieren nach PEA-XML" ausgewählt.

Vorbedingung:
Es muss (automatisch oder manuell) die CSP-OZ-DC-XML-Datei generiert worden sein.

Nachbedingung Erfolg:
Es existiert das PEA-XML.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Die CSP-OZ-DC-XML-Datei.

Anwendungsfall: Bilder exportieren

Akteure:
Nutzer

Ziel:
Ein Bild der UML-Darstellung generieren. Zum Beispiel als EPS.

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Exportieren als Bild" ausgewählt.

Vorbedingung:
UML-Darstellung muss vorhanden sein.

Nachbedingung Erfolg:
Die UML-Darstellung liegt in einem Bildformat vor.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Das aktuell im Arbeitsbereich angezeigte Diagramm.

Anwendungsfall: Drucken

Akteure:
Nutzer

Ziel:
Ausdruck der vorhandenen ausgewählten Daten

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Drucken" ausgewählt.

Vorbedingung:
Es müssen Daten zum Drucken ausgewählt sein.

Nachbedingung Erfolg:
Es liegt ein Ausdruck der ausgewählten Daten vor.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Das aktuell im Arbeitsbereich angezeigte Diagramm.

Anwendungsfall: Speichern (in XMI)

Akteure:
Nutzer

Ziel:
Datensicherung

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Speichern" ausgewählt.

Vorbedingung:
Daten zum Speichern müssen vorhanden sein.

Nachbedingung Erfolg:
Die Daten wurden erfolgreich gespeichert und sind wieder abrufbar.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Das UML-Modell.

Anwendungsfall: Laden (von XMI)

Akteure:
Nutzer

Ziel:
Darstellung und Bearbeitung vorher gesicherter Daten

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Laden" ausgewählt.

Vorbedingung:
Datei zum Laden muss vorhanden sein.

Nachbedingung Erfolg:
Die Daten wurden erfolgreich geladen und werden korrekt dargestellt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Die jeweilige Datei, die geladen werden soll.

Anwendungsfall: Neues Diagramm erstellen

Akteure:
Nutzer

Ziel:
Es soll ein neues Klassen- oder Strukturdiagramm erstellt werden.

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Klassendiagramm erstellen" oder „Strukturdiagramm erstellen" ausgewählt.

Nachbedingung Erfolg:
Es existiert das jeweilige Diagramm.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Diagrammname

Anwendungsfall: Klasse bearbeiten

Akteure:
Nutzer

Ziel:
Ändern der Daten einer Klasse

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Menü „Klasse bearbeiten" ausgewählt. Alternativ dazu ist auch ein Doppelklick auf die Klasse möglich.

Vorbedingung:
Es muss eine Klasse vorhanden sein.

Nachbedingung Erfolg:
Dialog zur Bearbeitung der Klasse erscheint. Die Änderungen werden nach Bestätigung des Dialogs übernommen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Daten für die Klasse

Anwendungsfall: Methode erstellen

Akteure:
Nutzer

Ziel:
Eine neue Methode erstellen.

Auslösendes Ereignis:
Der Nutzer markiert ein `capsule` oder ein `protocol` und wählt im Kontextmenü oder in der Symbolleiste „Methode erstellen".

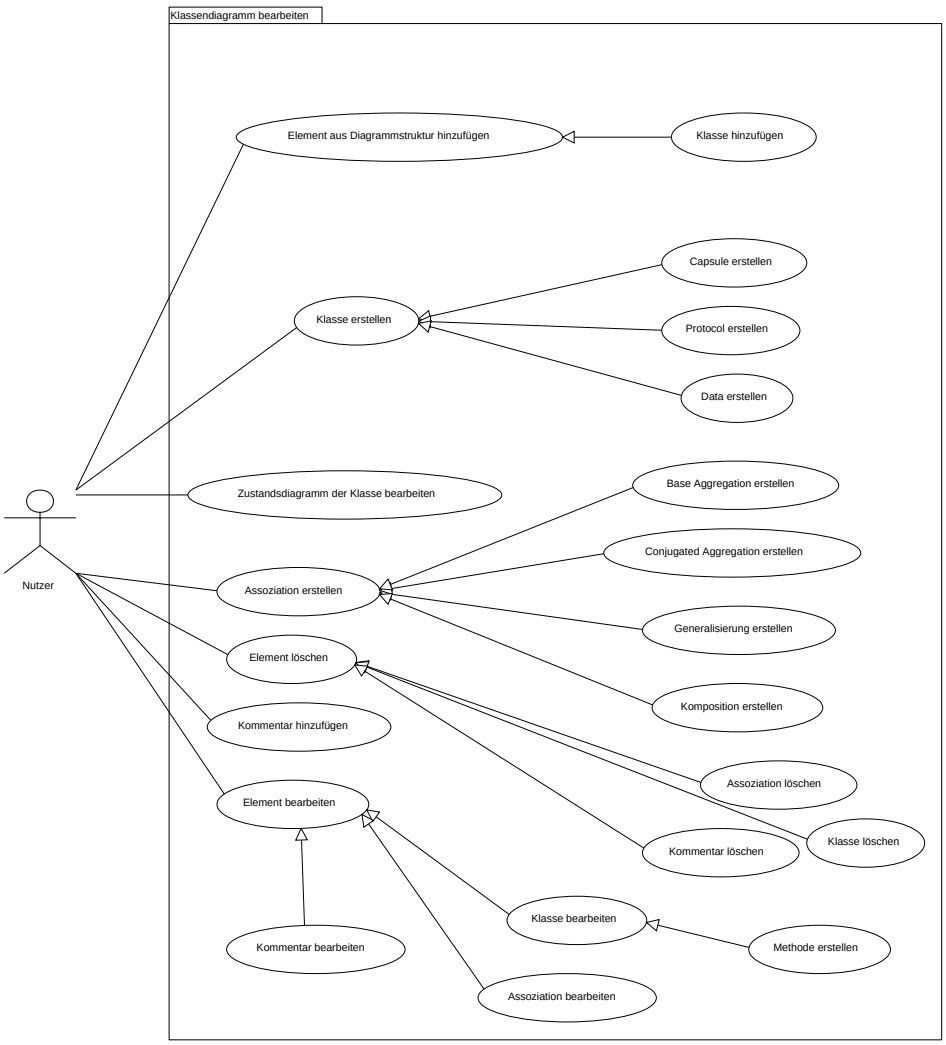
Vorbedingung:
Mindestens eine `capsule` oder ein `protocol` ist vorhanden.

Nachbedingung Erfolg:
Handelt es sich bei der ausgewählten Klasse um eine `capsule` wird die neue Methode zu dieser hinzugefügt. Wenn es sich bei dem markiertem Element um ein `protocol` handelt wird in den mit dem `protocol` verbundenen `capsules` und dem `protocol` die neue Methode erstellt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Ein ausgewähltes Element und ein Methodenname.

13.5.2 Anwendungsfälle in der Klassendiagrammansicht



Anwendungsfall: Klasse erstellen

Akteure:
Nutzer

Ziel:
Eine neue Klasse soll erstellt werden.

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Kontextmenü „Klasse erstellen" ausgewählt.

Vorbedingung:
-

Nachbedingung Erfolg:
Eine neue Klasse wird erzeugt und, wenn sie über die Symbolleiste erstellt wurde, dem Klassendiagramm hinzugefügt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Es muss der Name der Klasse und einer der folgenden drei Stereotypen gewählt werden: `capsule`, `protocol` und `data`.

Anwendungsfall: Zustandsdiagramm der Klasse bearbeiten

Akteure:
Nutzer

Ziel:
Für den CSP-Teil der Klasse wird ein zugehöriges Zustandsdiagramm zur Bearbeitung zur Verfügung gestellt.

Auslösendes Ereignis:
Der Benutzer wählt eine Klasse aus und klickt auf „Zustandsdiagramm der Klasse bearbeiten".

Vorbedingung:
Es muss mindestens eine Klasse vorhanden sein, zu der ein Zustandsdiagramm gewählt wird.

Nachbedingung Erfolg:
Zustandsdiagramms erscheint im Arbeitsbereich.

Nachbedingung Fehlschlag:
Fehlermeldung. Fenster zur Bearbeitung erscheint nicht.

Eingabedaten:
Eine ausgewählte Klasse.

Anwendungsfall: Assoziation erstellen

Akteure:
Nutzer

Ziel:
Eine Assoziation vom Typ Aggregation, Base-Aggregation, Conjugated-Aggregation, Komposition oder Generalisierung, soll zwischen zwei Klassen erstellt werden.

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Kontextmenü „Assoziation erstellen" ausgewählt. Eine Verbindung zwischen zwei Klassen wird per Mausklick gezogen.

Vorbedingung:
Mindestens zwei Klassen existieren, zwischen denen eine Assoziation erstellt werden kann.

Nachbedingung Erfolg:
Eine Assoziation wird dem Klassendiagramm und wenn nötig den zugehörigen Strukturdiagrammen hinzugefügt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Der Typ der Assoziation und bei Auswahl von Aggregation kann zusätzlich der Stereotyp `base` oder `conjugated` gewählt werden.

Anwendungsfall: Element löschen

Akteure:
Nutzer

Ziel:
Assoziation, Kommentar oder Klasse löschen

Auslösendes Ereignis:
Der Nutzer markiert ein Element und wählt im Kontextmenü oder in der Symbolleiste „Element löschen".

Vorbedingung:
Ein Element ist vorhanden.

Nachbedingung Erfolg:
Das Element verschwindet aus dem Klassendiagramm und falls kein Klassendiagramm dieses Element mehr enthält wird es aus dem UML-Modell entfernt. Zugehörige Strukturdiagrammen werde auf Konsistenz mit Klassendiagramm überprüft..

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Ein ausgewähltes Element

Anwendungsfall: Kommentar hinzufügen

Akteure:
Nutzer

Ziel:
Kommentar zur Klasse oder Assoziation hinzufügen.

Auslösendes Ereignis:
Der Nutzer markiert eine Klasse oder Assoziation und wählt im Kontextmenü oder in der Symbolleiste „Kommentar hinzufügen".

Vorbedingung:
Mindestens eine Klasse oder Assoziation existiert.

Nachbedingung Erfolg:
Ein Dialog zur Bearbeitung des Kommentars erscheint. Der Kommentar wird nach dem Bestätigen des Dialogs übernommen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Kommentartext

Alternative:
Anstatt eines Dialogs und der Auswahl von „Kommentar bearbeiten" kann die Darstellung auch direkt in der Oberfläche erfolgen.

Anwendungsfall: Element bearbeiten

Akteure:
Nutzer

Ziel:
Assoziation, Kommentar oder Klasse bearbeiten

Auslösendes Ereignis:
Der Nutzer markiert ein Element und wählt im Kontextmenü oder in der Symbolleiste „Element bearbeiten", oder der Nutzer macht einen Doppelklick auf das Element.

Vorbedingung:
Ein Element ist vorhanden.

Nachbedingung Erfolg:
Ein Dialog zur Bearbeitung des Elements erscheint. Die Änderungen werden nach dem Beenden des Dialogs übernommen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Ein ausgewähltes Element.

Alternative:
Anstatt eines Dialogs und der Auswahl von „Element bearbeiten" kann die Darstellung auch direkt in der Oberfläche erfolgen.

Anwendungsfall: Methode erstellen

Akteure:
Nutzer

Ziel:
Eine neue Methode erstellen.

Auslösendes Ereignis:
Der Nutzer markiert ein `capsule` oder ein `protocol` und wählt im Kontextmenü oder in der Symbolleiste „Methode erstellen".

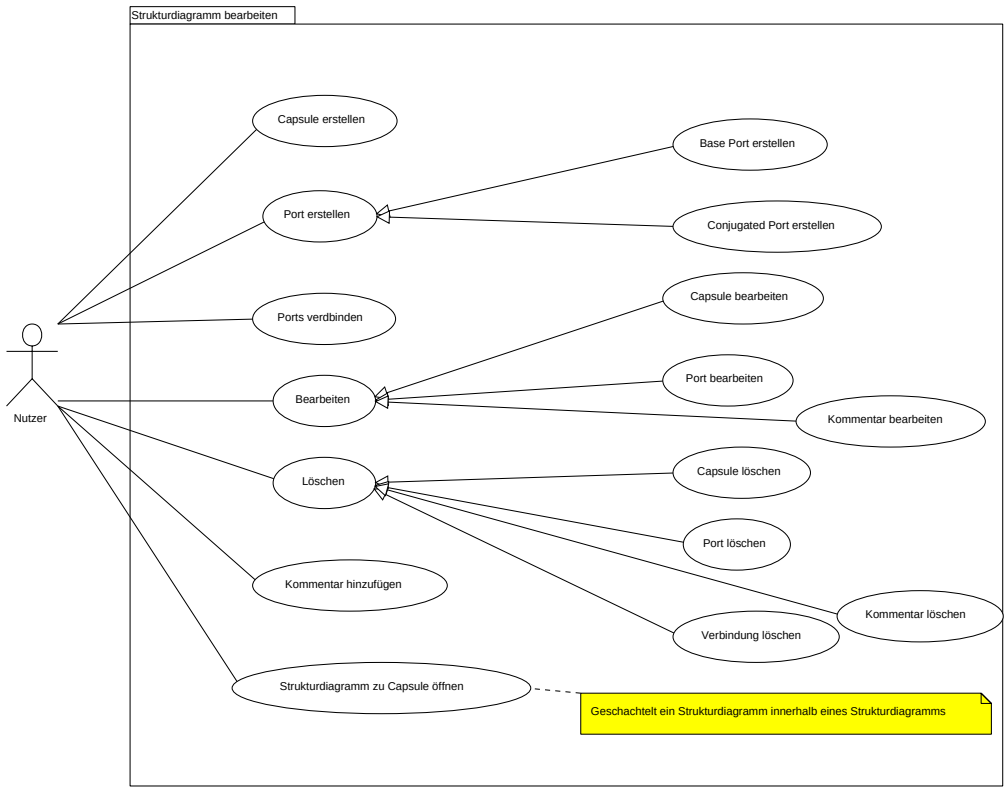
Vorbedingung:
Mindestens eine `capsule` oder ein `protocol` ist vorhanden.

Nachbedingung Erfolg:
Handelt es sich bei der ausgewählten Klasse um eine `capsule` wird die neue Methode zu dieser hinzugefügt. Wenn es sich bei dem markiertem Element um ein `protocol` handelt wird in den mit dem `protocol` verbundenen `capsules` und dem `protocol` die neue Methode erstellt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Ein ausgewähltes Element und einen Methodennamen.

13.5.3 Anwendungsfälle in der Strukturdiagrammansicht



Anwendungsfall: `capsule` erstellen

Akteure:
Nutzer

Ziel:
Eine neue `capsule` soll erstellt werden.

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Kontextmenü „`capsule` erstellen" ausgewählt.

Vorbedingung:
Ein Strukturdiagramm ist geöffnet.

Nachbedingung Erfolg:
Eine neue `capsule` wird in dem Strukturdiagramm hinzugefügt. Die neue `capsule` besitzt die zur ausgewählten Klasse gehörenden Ports.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Zu der `capsule` gehörende Klasse. Optional: Eine bereits ausgewählte `capsule`.

Anwendungsfall: Port erstellen

Akteure:
Nutzer

Ziel:
Eine neuer Port zu einer `capsule` soll erstellt werden.

Auslösendes Ereignis:
Der Nutzer hat entweder in der Symbolleiste oder im Kontextmenü „Port erstellen" ausgewählt.

Vorbedingung:
Ein Strukturdiagramm ist geöffnet. Eine `capsule` ist ausgewählt.

Nachbedingung Erfolg:
Eine neuer Port vom ausgewählten Typ wird der `capsule` hinzugefügt. Wurde ein neues Protokoll ausgewählt, so muss eine entsprechende neue Klasse mit Stereotyp `protocol` erstellt werden.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Vorhandenes oder neues Protokoll; Typ des Ports (`conjugated`/`base`); die ausgewählte `capsule`

mögliche Fehler:
Ungültiger Protokollname

Anwendungsfall: Ports verbinden

Akteure:
Nutzer

Ziel:
Zwei Ports sollen miteinander verbunden werden.

Auslösendes Ereignis:
Der Nutzer wählt das entsprechende Symbol in der Symbolleiste.

Vorbedingung:
Strukturdiagramm ist geöffnet. Mindestens zwei Ports sind verfügbar.

Nachbedingung Erfolg:
Die angegebenen Ports sind miteinander verbunden.

Nachbedingung Fehlschlag:
Eine Warn- bzw. Fehlermeldung wird an entsprechender Stelle angezeigt. Die Ports sind nicht miteinander verbunden.

Eingabedaten:
Die zwei zu verbindenen Ports

mögliche Fehler:
Die ausgewählten Ports stehen nicht über ein gemeinsames Protokoll in Beziehung.

Anwendungsfall: Bearbeiten

Akteure:
Nutzer

Ziel:
`capsule`, Port oder Kommentar bearbeiten.

Auslösendes Ereignis:
Der Nutzer markiert ein Element und wählt im Kontextmenü oder in der Symbolleiste „Element bearbeiten".

Vorbedingung:
Ein Element ist vorhanden.

Nachbedingung Erfolg:
Ein Dialog zur Bearbeitung des Elements erscheint. Die Änderungen werden nach dem Beenden des Dialogs übernommen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Zu bearbeitendes Element, zu ändernde Daten.

mögliche Fehler:
Bei Eingabe von Kardinalitäten sind alle Werte kleiner 1 ungültig.

Alternative:
Anstatt eines Dialogs und der Auswahl von „Element bearbeiten" kann die Darstellung auch direkt in der Oberfläche erfolgen.

Anwendungsfall: Element löschen

Akteure:
Nutzer

Ziel:
Eine `capsule`, ein Port, eine Verbindung zwischen Ports oder ein Kommentar sollen gelöscht werden.

Auslösendes Ereignis:
Der Nutzer markiert ein Element und wählt im Kontextmenü oder in der Symbolleiste „Element löschen".

Vorbedingung:
Mindestens ein Element ist im Diagramm vorhanden.

Nachbedingung Erfolg:
Das Element wird aus dem Strukturdiagramm gelöscht.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Das zu löschende Element; Möglicherweise eine Sicherheitsabfrage.

mögliche Fehler:
Es ist kein Element ausgewählt.

Anwendungsfall: Kommentar hinzufügen

Akteure:
Nutzer

Ziel:
Ein neuer Kommentar soll zu einer `capsule`, einem Port oder einer Verbindung hinzugefügt werden.

Auslösendes Ereignis:
Der Nutzer markiert ein Element im Diagramm und wählt im Kontextmenü oder in der Symbolleiste „Kommentar hinzufügen" aus.

Vorbedingung:
Es existiert ein Element.

Nachbedingung Erfolg:
Dem ausgewählten Element wird ein Kommentar hinzugefügt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Element, zu dem der Kommentar hinzugefügt werden soll, der Kommentartext.

mögliche Fehler:
Es ist kein Element ausgewählt.

Anwendungsfall: Strukturdiagramm zu `capsule` öffnen

Akteure:
Nutzer

Ziel:
Das Strukturdiagramm zu einer `capsule` soll geöffnet werden.

Auslösendes Ereignis:
Der Nutzer wählt im Kontextmenü oder in der Symbolleiste den entsprechenden Eintrag.

Vorbedingung:
Eine `capsule` existiert.

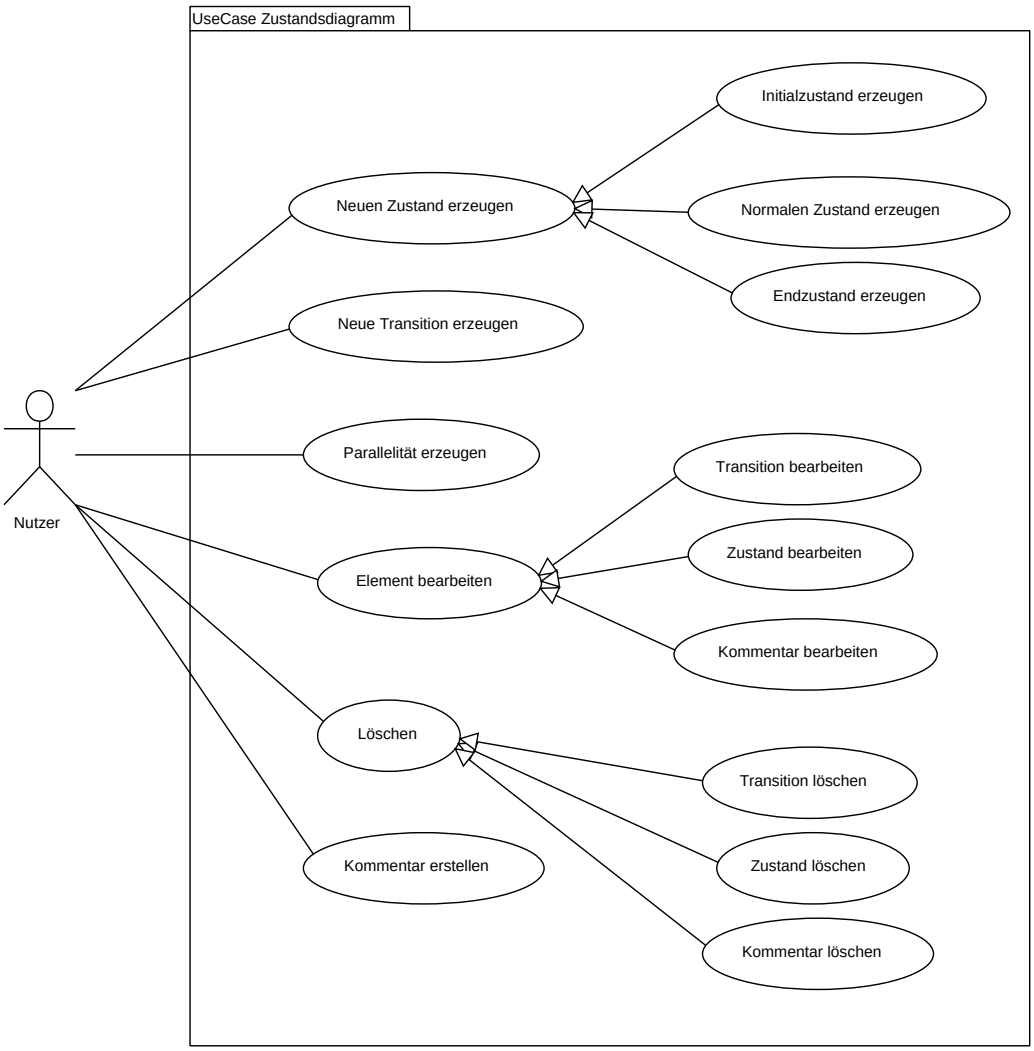
Nachbedingung Erfolg:
Das Strukturdiagramm für das Innere der `capsule` wird angezeigt. Existierte bisher kein Diagramm, so wird ein neues erzeugt und angezeigt. Dieses beinhaltet capsules, welche als Teil in einer Komposition mit der äusseren capsule in Verbindung stehen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
`capsule` zu der das Diagramm angezeigt werden soll.

mögliche Fehler:
Es ist keine `capsule` ausgewählt.

13.5.4 Anwendungsfälle in der Zustandsdiagrammansicht



Anwendungsfall: Neuen Zustand erzeugen

Akteure:
Nutzer

Ziel:
Ein neuer Initial-, Normal- oder Endzustand soll erzeugt werden.

Auslösendes Ereignis:
Der Nutzer wählt den entsprechenden Eintrag im Kontextmenü oder in der Symbolleiste aus.

Vorbedingung:
Zustandsdiagramm ist geöffnet.

Nachbedingung Erfolg:
Der Zustand wird erzeugt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Typ des Zustands

Anwendungsfall: Neue Transition erzeugen

Akteure:
Nutzer

Ziel:
Eine neue Transition zwischen zwei Zuständen soll erzeugt werden.

Auslösendes Ereignis:
Der Nutzer wählt die entsprechende Funktion im Kontextmenü oder der Symbolleiste aus.

Vorbedingung:
Es existieren mindestens zwei Zustände.

Nachbedingung Erfolg:
Eine neue Transition wird erzeugt. Falls eine Methode ausgewählt ist, so muss diese an der Transition erscheinen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Zu verbindende Zustände; Optional: zur Transition gehörende Methode

Anwendungsfall: Superzustand erzeugen

Akteure:
Nutzer

Ziel:
Es soll ein Superzustand erzeugt werden.

Auslösendes Ereignis:
Der Nutzer wählt die entsprechende Funktion im Kontextmenü oder in der Symbolleiste aus oder der Nutzer zieht einen oder mehrere Zustände in einen bestehenden Zustand hinein (Drag and Drop).

Vorbedingung:
Es existiert mindestens ein Zustand (zwei bei Drag and Drop).

Nachbedingung Erfolg:
Ein Superzustand wird erzeugt. Die ausgewählten Zustände werden in den Superzustand eingefügt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Ausgewählte Zustände, bei Drag and Drop: Zielzustand

Mögliche Fehler:
Es ist kein Zustand ausgewählt.

Anwendungsfall: Parallelität erzeugen

Akteure:
Nutzer

Ziel:
In einem Zustand soll Parallelität im Sinne von CSP erzeugt werden.

Auslösendes Ereignis:
Der Nutzer wählt die entsprechende Funktion im Kontextmenü oder in der Symbolleiste aus.

Vorbedingung:
Ein Zustand existiert.

Nachbedingung Erfolg:
Der ausgewählte Zustand wird zu einem Zustand mit Parallelität.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Ausgewählter Zustand.

Mögliche Fehler:
Es ist kein Zustand ausgewählt.

Anwendungsfall: Element bearbeiten

Akteure:
Nutzer

Ziel:
Zustand, Transition oder Kommentar bearbeiten.

Auslösendes Ereignis:
Der Nutzer markiert ein Element und wählt im Kontextmenü oder in der Symbolleiste „Element bearbeiten".

Vorbedingung:
Ein Element ist vorhanden.

Nachbedingung Erfolg:
Ein Dialog zur Bearbeitung des Elements erscheint. Die Änderungen werden nach dem Beenden des Dialogs übernommen.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Zu bearbeitendes Element, zu ändernde Daten

Mögliche Fehler:
Ungültige Daten werden eingegeben.

Alternative:
Anstatt eines Dialogs und der Auswahl von „Element bearbeiten" kann die Darstellung auch direkt in der Oberfläche erfolgen.

Anwendungsfall: Element löschen

Akteure:
Nutzer

Ziel:
Ein Zustand, eine Transition oder ein Kommentar soll gelöscht werden.

Auslösendes Ereignis:
Der Nutzer markiert ein Element und wählt im Kontextmenü oder in der Symbolleiste „Element löschen".

Vorbedingung:
Mindestens ein Element ist im Diagramm vorhanden.

Nachbedingung Erfolg:
Das Element wird aus dem Strukturdiagramm gelöscht. Falls nötig werden alle abhängigen Elemente gelöscht.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Das zu löschende Element.

Mögliche Fehler:
Es ist kein Element ausgewählt.

Anwendungsfall: Kommentar hinzufügen

Akteure:
Nutzer

Ziel:
Ein neuer Kommentar soll zu einem Zustand oder einer Transition hinzugefügt werden.

Auslösendes Ereignis:
Der Nutzer markiert ein Element im Diagramm und wählt im Kontextmenü oder in der Symbolleiste „Kommentar hinzufügen" aus.

Vorbedingung:
Es existiert ein Element.

Nachbedingung Erfolg:
Dem ausgewählten Element wird ein Kommentar hinzugefügt.

Nachbedingung Fehlschlag:
Eine aussagekräftige Fehlermeldung wird ausgegeben.

Eingabedaten:
Element, zu dem der Kommentar hinzugefügt werden soll und der Kommentartext.

mögliche Fehler:
Es ist kein Element ausgewählt.

Kapitel 14

Modifizierung der Anforderungen

14.1 Einleitung

In diesem Kapitel werden die Änderungen beschrieben, die sich im Laufe des Projektes in Bezug auf die Anforderungen ergeben haben. Hauptsächlich geht es dabei um Änderungen des UML-Profils. Aus diesem Grund werden die Anpassungen des UML-Profil ausführlich im folgenden Abschnitt beschrieben.
Andere Änderungen sind im Abschnitt 14.3 *Sonstige Änderungen*, S. 145 aufgeführt.

14.2 Anpassung des UML-Profils

Da die *Object Management Group* vor Beginn des Projektes die Version 2.0 der UML verabschiedet hat, das vorhandene Profil jedoch noch auf der Version 1.5 basierte, soll hier eine Anpassung auf die aktuelle Version erfolgen. Hierbei wurde darauf geachtet, dass sich das neue Profil in großen Teilen wie das alte benutzen lässt.
Die Änderungen beziehen sich größtenteils auf die Strukturdiagramme, die aus der UML-RT übernommen wurden. Diese konnten komplett durch die in der UML 2.0 überarbeiteten Komponentendiagramme ersetzt werden, welche eine ähnliche Bedeutung besitzen. Der Stereotyp *capsule* ist nun eine Erweiterung des in der UML definierten Stereotyps *component*. Die *capsule*s kommunizieren über festgelegte Schnittstellen (*interface*s) miteinander. Diese werden entweder von einer *capsule* benötigt oder bereitgestellt. Die genauen Unterschiede werden nun anhand des Fallbeispiels *Bank*, das schon in Abschnitt 9 (S. 77) vorgestellt wurde, dargestellt.
Die Abbildungen 14.1, 14.2 und 14.3 stellen die Diagramme nach dem alten Profil denen des neuen Profils gegenüber. Das Verhalten der Klassen wird weiterhin über Zustandsautomaten spezifiziert, an deren Notation sich nichts geändert hat.
Die stärksten Veränderungen der Klassendiagrammnotation betrifft die Anbindung von *capsule*s an Schnittstellen, welche im alten Profil durch Klassen mit dem Stereotyp *protocol* dargestellt wurden. Da im Laufe der Umstellung des Profils auf die UML 2.0 festgestellt wurde, dass die gewünschte Sematik dieser Klassen mit der Semantik des UML-Stereotyps *interface* übereinstimmt, wird nun eine Schnittstelle mit diesem Stereotyp markiert. Die Verbindung der *capsule*s mit den Schnittstellen wurde im alten Profil über Aggregationen mit den Stereotypen *base* und *conjugated* modelliert. Durch

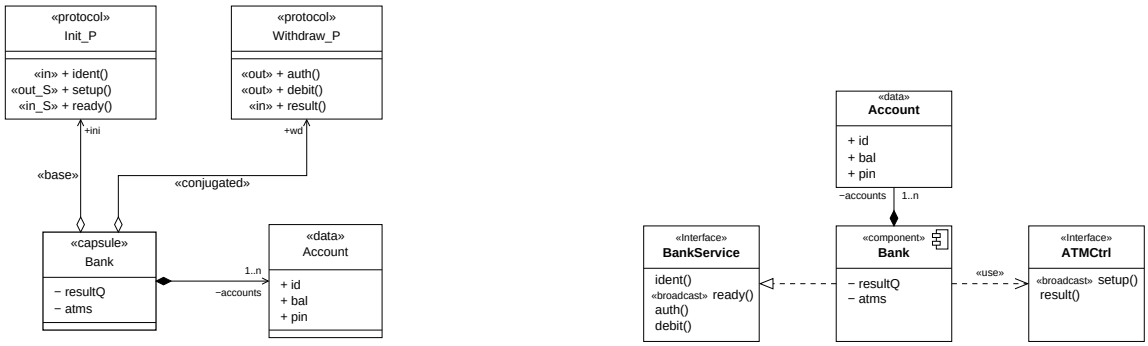


Abbildung 14.1: Klassendiagramm für die Bank nach dem alten/neuen UML-Profil

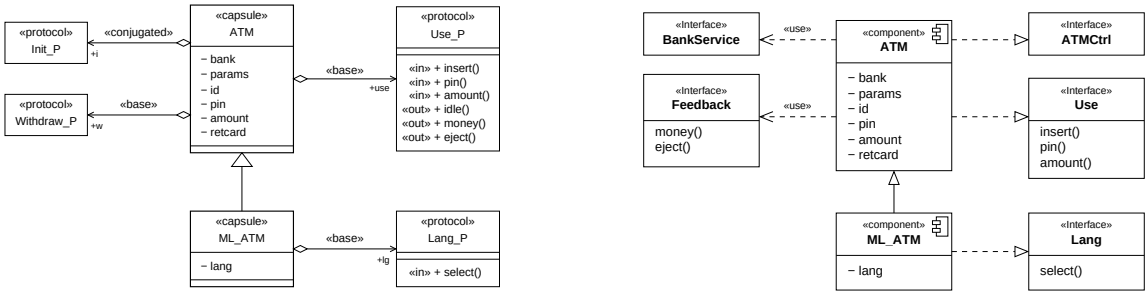


Abbildung 14.2: Klassendiagramm für die ATM nach dem alten/neuen UML-Profil

die Umstellung des Stereotyps der Schnittstellen bot sich auch eine neue Modellierung dieser Verbindungen an, indem Basiselemente der UML genutzt werden. Namentlich sind dies Assoziationen mit den Sterotypen *realization* und *dependency*, welche auch ein eigenes Notationselement – den gestrichelten Pfeil mit einem weißen Dreieck – beziehungsweise mit einem Winkel als Pfeilspitze, besitzen. Die *realization* entspricht hierbei einer Aggregation mit dem Stereotyp *base*, die *dependency* entspricht einer Aggregation mit dem Stereotyp *conjugated*.
Im alten Diagramm war es möglich, mit den Sterotypen *in* und *out* innerhalb einer Schnittstelle zwei „Richtungen“ der Kommunikation zu definieren. Um die Bedeutung der neuen Verbindungen innerhalb der UML beizubehalten, wurden diese beiden Stereotypen aus dem Profil entfernt, so dass alle Methoden einer Schnittstelle, die von einer *capsule* „realisiert“ wird, nun als CSP-Kanäle der Form `method` übersetzt werden. Entsprechend werden alle Methoden einer Schnittstelle, von der eine *capsule* „abhängig“ ist, in Kanäle des Typs `chan` umgewandelt. Da weiterhin eine Unterscheidung zwischen „eins zu eins“- und „viele zu viele“-Kommunikation nötig ist, wurden die Stereotypen *simple* und *broadcast* hinzugefügt.
Um eine UML 2.0 konforme Modellierung der Instanziierung des zu spezifizierenden Systems zu erhalten, wurden die Strukturdiagramme aus der UML-RT durch Komponentendiagramme ersetzt (Abbildung 14.3). Diese wurden mit der Version 2.0 in die UML mit aufgenommen. Die Änderungen gegenüber der UML-RT sind nicht gravierend, sondern beziehen sich weitestgehend auf die Notationselemente und nicht deren

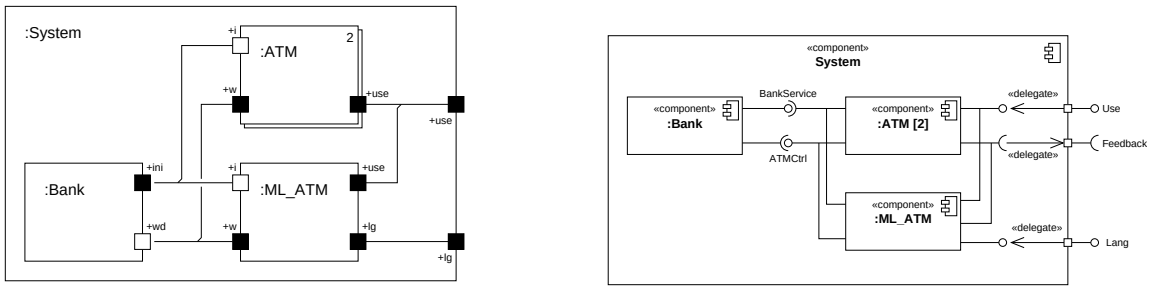


Abbildung 14.3: Strukturdiagramm/Komponentendiagramm des Systems nach dem alten/neuen UML-Profil

Bedeutung. So werden nun Verbindungen über eine Schnittstelle nicht mehr anhand der Farbe des Ports, sondern über weitere Notationselemente der UML dargestellt. Diese Elemente sind das „implemented interface“, welches mittels eines Kreises, der mit einer Linie mit der assoziierten Komponente verbunden ist, dargestellt wird, beziehungsweise das „required interface“, welches anstelle eines Kreises einen Halbkreis besitzt. Die Kommunikation zweier Komponenten wird über einen sogenannten *assembly connector* modelliert, der einen Kreis darstellt, der von einem Halbkreis umfasst wird. Wie auch im alten Profil, ist es möglich, dass mehrere Komponenten über dieselbe Schnittstelle miteinander kommunizieren, und sich auf dieser Kommunikation synchronisieren, also dass alle verbundenen Komponenten gleichzeitig die entsprechende Methode ausführen. Hierzu werden alle entsprechenden Komponenten über denselben *assembly connector* verbunden.

14.3 Sonstige Änderungen

Bei der folgenden Aufzählung wird immer zu Beginn auf den Bezugspunkt in Produktskizze oder Anforderungsdefinition hingewiesen. Wenn gewünschte Funktionalität zwar bedacht aber nicht umgesetzt wurde, so wird dies im Kapitel *Ausblick* (S. 209) besprochen.

- Produktskizze, 12.2: Es gibt keine speziellen Buttons für den Export. Dieser ist stattdessen über das Hauptmenü bzw. das Kontextmenü erreichbar.
- Produktskizze, 12.2.2: Es ist nicht möglich, in einem Komponentendiagramm ein neues Komponentendiagramm für das Spezifizieren des Inneren einer *capsule* anzulegen. Das Innere eine *capsule* kann direkt in der *capsule* in genau einem der Komponentendiagramme spezifiziert werden, in denen die *capsule* auftaucht.
- Produktskizze, 12.2.3: Es gibt keinen Button oder Menüeintrag *Superzustand erzeugen*, die Funktionalität ist nur über *Drag and Drop* möglich.
- Produktskizze, 12.2.3: Parallelität wird über UML-Regions ermöglicht.
- Produktskizze, 12.3: Das Laden und Speichern erfolgt nicht mithilfe von XMI sondern mit einem eigenen XML-Format.

- Anforderungsdefinition, 13.2: Eine Druckfunktion muss nicht vorhanden sein, da voraussichtlich in den meisten Fällen die Grafiken in einen Textkontext gestellt werden sollen. Deswegen ist die Bildexport-Funktion ungemein wichtiger als eine Druckfunktion. Außer als PNG und EPS ist es möglich, BMP und JPEG-Bilder zu exportieren.

- Anforderungsdefinition, 13.2.2, Object-Z-Datentypen und -Ausdrücke und DC-Ausdrücke, 2.: Der DC-Formeleditor ist nur für Countertraces und nicht für Testformeln vorgesehen, da es im CSP-OZ-DC nur Countertraces gibt.

- Anforderungsdefinition, 13.2.3, Allgemeine Eigenschaften, 9. d): Es ist gar nicht möglich, Transitionen mit Operationen zu beschriften, die in keiner Klasse definiert sind, deswegen wird dieser Vorgang auch nicht als Fehler angezeigt.

- Produktskizze und Anforderungsdefinition allgemein: Alles was in diesen Dokumenten über Kommentare und sogenannte Metadaten steht, wird durch folgendes ersetzt: Es soll zum einen Kommentare für Elemente geben, die in einer Property-Ansicht geschrieben werden können. Desweiteren soll es optional möglich sein, „Notizzettel“ direkt in Diagramme zu zeichnen und mit einer gestrichelten Linie mit anderen Diagrammelementen zu verbinden.

Kapitel 15

Entwurf

15.1 Einleitung

Dieses Kapitel beschreibt den Entwurf von *Syspect*, also die Modellierung eines Software-Produkts zur Umsetzung der Anforderungsdefinition. Üblicherweise werden dabei zwei Ebenen unterschiedlichen Abstraktionsgrades unterschieden: Die Architektur bzw. der Grobentwurf beschreibt die wesentlichen Komponenten der Software sowie die Kommunikation dieser Komponenten über Schnittstellen. Hierauf baut dann der Feinentwurf auf, der möglichst präzise die einzelnen Komponenten modelliert und somit als Implementierungsgrundlage für die Entwickler dienen kann.
Während in der Architektur wichtige Entwurfsentscheidungen erläutert werden, wird im Feinentwurf aufgezeigt, wie ihre positiven Konsequenzen genutzt und die vereinzelt auch auftretenden negativen Konsequenzen abgefangen werden können.

Da wir uns entschieden haben, auf der *Eclipse*-Plattform aufzubauen, übernehmen wir die hier vorliegende Architektur und das dafür wesentliche Plug-in-Konzept. Die *Eclipse*-Plattform selbst, verwendete Plug-ins, so wie eigene als Plug-ins entwickelte Komponenten werden in Abschnitt 15.2 beschrieben und ihr Zusammenwirken im Gesamtsystem *Syspect* grob erläutert.
Im Abschnitt 15.3 hingegen wird die Funktionsweise jedes einzelnen Plug-ins sowie seine bereitgestellten und benötigten Schnittstellen und spezielle verwendete Prinzipien erklärt.

Schließlich ist ein weiteres Ziel dieses Kapitels, die Wiederverwendung und Weiterentwicklung von *Syspect* zu ermöglichen und zu erleichtern. Deswegen wurde Wert darauf gelegt, die Schnittstellen so einfach wie möglich zu halten und so das Austauschen vorhandener Komponenten sowie das Hinzufügen neuer Komponenten komfortabel zu gestalten.

15.2 Architektur

Da viele der funktionalen Anforderungen bereits in freier Software umgesetzt worden sind, wurde beschlossen, keine Applikation von Grund auf neu zu entwerfen, sondern

vorhandene Software zu nutzen. Nach kurzer Produktanalyse (siehe Kapitel 1, S. 7, Kapitel 2, S. 17, Kapitel 3, S. 27 und Kapitel 4, S. 33) wurde die *Eclipse*-Plattform [EFI05b] zusammen mit einigen *Eclipse*-Plugins gewählt:

- Das *Graphical Editing Framework* (GEF) wird für die Erstellung grafischer Editoren verwendet.
- Das *Graphical Modeling Framework* (GMF) wird für den Image-Export verwendet.

Weiterhin werden von *Syspect* das PEA-Tool von Jochen Hoenicke und Roland Meyer für den Export von PEA-XML und die *Community Z Tools* für die Eingabe, Verarbeitung und Datenhaltung von Z-Ausdrücken (Siehe Kapitel 8, S. 69 und Kapitel 5, S. 43) verwendet. Abbildung 15.1 stellt grob die Abhängigkeiten dieser verschiedenen Softwarekomponenten dar.

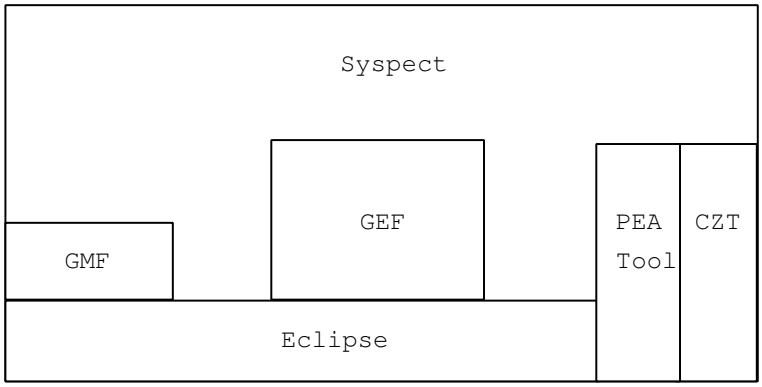


Abbildung 15.1: Abhängigkeiten des Syspect Werkzeuges

Die Architektur von *Syspect* kann also keineswegs als allein stehend betrachtet werden, sondern muss im Kontext der bestehenden Software gesehen werden, in die sie eingebettet ist.
Im folgenden Unterabschnitt wird deswegen zunächst dieser Kontext und seine wesentlichen Komponenten näher beschrieben.

15.2.1 Umgebung und Einbettung

Eclipse besteht zunächst aus der *Eclipse*-Plattform, welche ein Laufzeitmodul ist, das Plug-ins verwalten kann. Alle weiteren Funktionalitäten werden durch Plug-ins bereitgestellt. Zusätzliche Flexibilität wird dadurch erreicht, dass Plug-ins (wie das weiter unten beschriebene GEF) auch Schnittstellen für andere Plug-ins bereitstellen können. Die Verwendung als Entwicklungsumgebung ist die gängigste Möglichkeit *Eclipse* einzusetzen. *Eclipse* stellt weiterhin eine *Rich Client Platform* (RCP) zur Verfügung, mit der es möglich ist, ein eigenes Softwareprodukt auf der Basis des *Eclipse*-Frameworks zu erstellen, in dem dann nur die dafür benötigten Plug-ins enthalten sind.
Durch dieses Konzept lässt sich, trotz der anfangs etwas längeren Einarbeitungszeit, sehr viel Zeit bei der Implementierung häufig genutzter Komponenten wie Wizards und

Editoren einsparen. Dies ist auch deswegen besonders vorteilhaft, weil die Erweiterbarkeit nicht durch die bereitgestellten Komponenten beeinträchtigt wird, da genügend Schnittstellen für eigene Anpassungen und Erweiterungen zur Verfügung stehen.
Aufgrund dieser Kombination von bereitgestellter Funktionalität und flexibler Erweiterbarkeit, sowie der großen Entwicklergemeinde – und damit vorhandene Dokumentation, Stabilität und Erprobtheit – haben wir uns dafür entschieden, *Syspect* als Plug-in für Eclipse bzw. als RCP-Applikation zu entwickeln.

GEF und Draw2D

Für die grafischen Editoren von *Syspect* wird das Grafik-Framework und *Eclipse*-Plug-in GEF verwendet. Dieses realisiert eine *Model-View-Controller*-Architektur (MVC) [eNo06], d.h. es trennt die Ansicht (*View*) von der Datenhaltung (*Model*).
Die *View* wird bei GEF durch so genannte *Figures* gebildet, welche die zu zeichnenden Elemente in einem Diagramm darstellen. *Figures* werden aus dem GEF-Framework bzw. im Detail aus der *Draw2D*-Komponente abgeleitet. Die *Draw2D*-Komponente baut auf SWT(siehe Kapitel 2, S. 17) auf, wobei diese beiden Komponenten auch außerhalb von *Eclipse* als Grafik-Bibliotheken verwendet werden können. Aufgrund dieser Bibliotheken werden die *Figures* von *Syspect* unabhängig von anderen Komponenten und sogar unabhängig von *Eclipse* sein. Zudem stellt *Draw2D* mächtige Layout-Algorithmen für gerichtete Graphen zur Verfügung.
Das Framework stellt an das *Model* keine Anforderungen. Allgemein wird es jedoch meist ein *Model*-Objekt geben, das ein *Figure*-Objekt repräsentiert.

Der *Controller* wird bei GEF durch so genannte *Editparts* gebildet. Die *Editparts* sind die Verbindungsstücke zwischen dem *Model* und der *View*, wobei es hier in der Regel ein *Editpart* für ein *Model*- und ein *Figure*-Objekt gibt. Die *Editparts* sind an das *Model* nur lose gekoppelt, was durch ein *Factory*-Pattern [GHJ95] und *Observer*-Pattern [GHJ95] realisiert wird. Die *Factory* erstellt anhand eines *Model*-Objekts den zugehörigen *Editpart* und der *Editpart* erstellt die zugehörige *Figure*. Diese Zusammenhänge sind in Abbildung 15.2 mit den durchgezogenen Pfeilen dargestellt.

Für die *Event*-Behandlung stellt GEF ebenfalls ein Entwurfsmuster bzw. Schnittstellen zur Verfügung. *Events* werden in so genannten *Editpolicies* gekapselt, die einem *Editpart* zugeordnet werden. Benutzereingaben von der GUI werden dann als *Request* zu den entsprechenden *Editpolicies* weitergeleitet. Hier gibt es für alle möglichen *Events* eine entsprechende *Editpolicy*. Für die Verarbeitung der *Events* werden von den *Editpolicies* so genannte *Commands* erstellt, die Änderungen an dem *Model* durchführen. Ein *Command* bietet standardmäßig Unterstützung für *Undo*- und *Redo*-Funktionalitäten. Die Änderungen an dem *Model* werden über das Observer-Pattern an den entsprechenden *Editpart* weitergeleitet, der schließlich die *Figure* entsprechend aktualisiert. Diese Zusammenhänge sind in Abbildung 15.2 mit den gestrichelten Pfeilen dargestellt.

Übersicht über die in GEF verwendeten Entwurfsmuster/Architekturmuster [EFI06d]

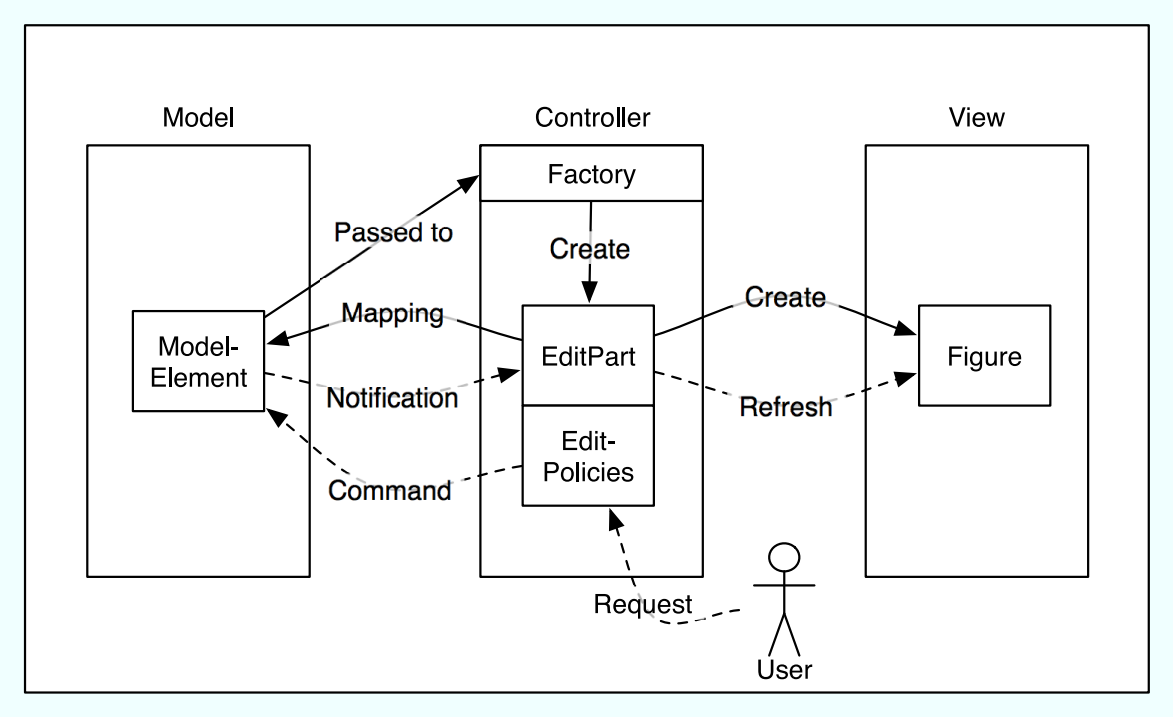


Abbildung 15.2: Wesentliche Abläufe des GEF-Frameworks

- Das *Model-View-Controller*-Muster wird verwendet um Schnittstelle, Verhalten und Präsentation von einander zu entkoppeln. Jedes Java-Objekt kann ein *Model*-Element sein. Jedes *View*-Element muss das Interface *IFigure* implementieren.

- Das *Command*-Muster wird verwendet, um Änderungen am *Model* zu kapseln.

- Das *Chain of Responsibility*-Muster entkoppelt Sender von *Requests* von den Empfängern, in dem mehr als einem Empfänger die Möglichkeit gegeben wird, den *Request* zu verarbeiten. Zum Beispiel gibt es *Editpolicies* die *Requests* an mehrere verkettete *Commands* weitergegeben.

- Das *State*-Muster erlaubt dem Editor, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Zum Beispiel verhält sich der Editor bei einem ausgewählten *marquee tool* anders, als wenn ein *create tool* aktiv ist.

- Das *Abstract Factory*-Muster stellt eine Schnittstelle für das Erstellen von Familien mit einer in Beziehung stehender Objekte zur Verfügung. Dies kann in GEF zum Erstellen von *Editparts* verwendet werden.

Das GEF-Framework stellt sowohl eine Bandbreite an Editor-Funktionalität zur Verfügung, die von *Syspect* genutzt oder angepasst werden kann, als auch eine solide, modulare Architektur.

GMF

GMF Das *Graphical Modeling Framework* ist ein Plug-in für das Generieren von grafischen Editoren und basiert auf GEF und EMF (siehe auch Kapitel 16.1, S. 209). Da GMF jedoch ein sehr neues, unerprobtes Produkt ist, konnten wir es leider nicht für die Generierung unserer grafischen Editoren nutzen.
Das einzige, was wir von diesem Plug-in für uns verwendet haben, ist einige Funktionalität für den Image-Export (siehe auch Abschnitt 15.3.15, S. 201).

PEA-Tool

Das PEA-Tool [HM05] ist ein Software-Werkzeug von Jochen Hoenicke und Roland Meyer und dient dazu, Phasen-Event-Automaten (PEA) in eine Java-Repräsentation darzustellen. Das Werkzeug bietet darüber hinaus die Möglichkeiten, PEA nach XML zu konvertieren, DC-Countertrace-Formeln in PEA umzuwandeln und das Parallel-Produkt von PEAs zu berechnen. Bislang fehlte eine Möglichkeit, die Transitionen in PEA mit Z-Ausdrücken zu beschriften, diese wurde im Laufe unseres Projektes hinzugefügt. Die XML-Darstellung, die vom PEA-Tool produziert wird, kann als Eingabe für einen Model-Checker benutzt werden.
Intern gliedert sich das PEA-Tool folgendermaßen: Die Klasse `PhaseEventAutomata` dient als Repräsentation von PEA. Die Phasen werden durch Objekte der Klasse `Phase` beschrieben und die Transitionen durch Objekte der Klasse `Transition`. Formeln, die für Invarianten oder Guards verwendet werden, werden als *Constraint Decision Diagram* (CDD) durch die gleichnamige Klasse dargestellt. Die CDDs lassen sich durch verschiedene *Decision*-Klassen einfach generieren. Formeln des Duration Calculus lassen sich mit Hilfe der Klasse `CounterTrace` in PEA umwandeln. Ein weiterer Teil des PEA-Tools ist das Package `modelchecking`. Hier finden sich verschiedene Compiler und Konverter zur Umwandlung von Formeln in bestimmte Darstellungen, wie z.B. DNF oder zur Konvertierung in eine XML-Darstellung. Die wichtigste Klasse für uns stellt hier die Klasse `PEA2XMLConverterFast` dar. Sie wandelt `PhaseEventAutomata`-Objekte in eine XML-Darstellung um und produziert direkt den XML-Output.

Das PEA-Tool wurde als *Eclipse*-Plug-in in unsere Architektur integriert.

Community-Z

Community-Z-Tools (CZT) ist ein Projekt, das eine Sammlung von Werkzeugen für die Spezifikationssprache Z und teilweise auch Object-Z bereitstellt.

Folgende Funktionalitäten von CZT werden von uns verwendet:

- Die Entwicklung des XML-Schemas *ZML* ermöglicht eine strukturierte Speicherung von Z-Spezifikationen. Dieses Schema kann von uns verwendet und erweitert werden, um CSP-OZ-DC-Spezifikationen als XML zu speichern.
- CZT bietet mit den Werkzeugen *CoreJava* und *GnAST* die Möglichkeit, Z-Ausdrücke als Java-Klassen zu speichern. Mit Hilfe des CZT-*Parsers* und dieser

Werkzeuge wird das Umwandeln von ZML, LaTeX-Ausdrücken und Unicode in Java-Klassen und -Interfaces sowie auch umgekehrt ermöglicht.

- Ebenfalls im Projekt enthalten ist ein *Type-Checker* zum Überprüfen von Z-Ausdrücken auf Typkorrektheit.

Das Verwenden des Speicherformats *ZML* sowie das Parsen und Konvertieren der verschiedenen Formate konnte von uns gut weiterverwendet werden und reduzierte somit den Arbeitsaufwand.
Die Funktionalitäten, die wir von CZT verwenden wollen, sind in unserem Plug-in *OZ-DC-Util* (siehe Abschnitt 15.3.11, S. 189) integriert.

15.2.2 Plug-in-Struktur

Aufbauend auf dem *Plug-in*-Konzept von *Eclipse* haben wir uns für einen sehr modularen Ansatz entschieden, unser Programm zu gliedern. Ein *Plug-in* ist hierbei eine Softwarekomponente, deren Einsatzmöglichkeiten von ihrer Schnittstellendefinition abhängt. Dort wird festgelegt, wie das Plug-in innerhalb einer Applikation verwendet werden kann, von welchen anderen Plug-ins es abhängig ist bzw. welche Funktionalität es nach außen hin bereitstellt.

In *Eclipse* werden zwei Arten von Abhängigkeiten unterschieden:

- Einfache *Dependencies* liegen dann vor, wenn eine Komponente A von Komponente B abhängt, ohne das der gegenseitige Fall zutrifft.

- Bei einer wechselseitigen Abhängigkeit müssen im Gegensatz dazu sogenannte *Extension Points* eingesetzt werden. D.h. wenn Komponente A von Komponente B und andersherum abhängig ist, dann wäre eine Möglichkeit, zu definieren, A stellt *Extension Points* zur Verfügung, die von B implementiert werden. B behält seine Abhängigkeit zu A.

Im Folgenden soll ein Überblick über das Zusammenwirken der *Syspect*-Plug-ins gegeben werden. Der zentrale Einstiegspunkt ist dabei `de.syspect.core`. Erläuterungen zu den einzelnen Plug-ins sind im Kapitel „Feinentwurf“ (Abschnitt 15.3, S. 157) zu finden.

de.syspect.core Das Core-Plug-in implementiert die *Extension Points* der Rich Client Platform und stellt darüberhinaus die Verbindung zwischen bestimmten Plug-ins her. Es definiert eine Reihe eigener *Extension Points*, z.B. für das *Model*. Die anderen Plug-ins können nun auf die bereitgestellten Funktionen des Core-Plug-ins zugreifen. Aus Gründen der Übersichtlichkeit ist der Großteil der weiteren Funktionalität in andere Plug-ins ausgelagert. Dabei unterscheiden wir zwei grundsätzliche Arten:

Library-Plug-ins Library-Plug-ins implementieren in der Regel keine vollständige Funktionalität. Sie dienen als Ergänzungen der anderen Plug-ins. So gibt es z.B. `de.syspect.editorbundle` das von den Editoren benutzt wird, um gemeinsam genutzten Code geordnet zu verwalten. In dem Editor-Bundle befinden sich beispielweise eine Reihe abstrakter Commands.
Die Abhängigkeiten von den Library-Plug-ins sind in Abbildung 15.3 zu sehen.

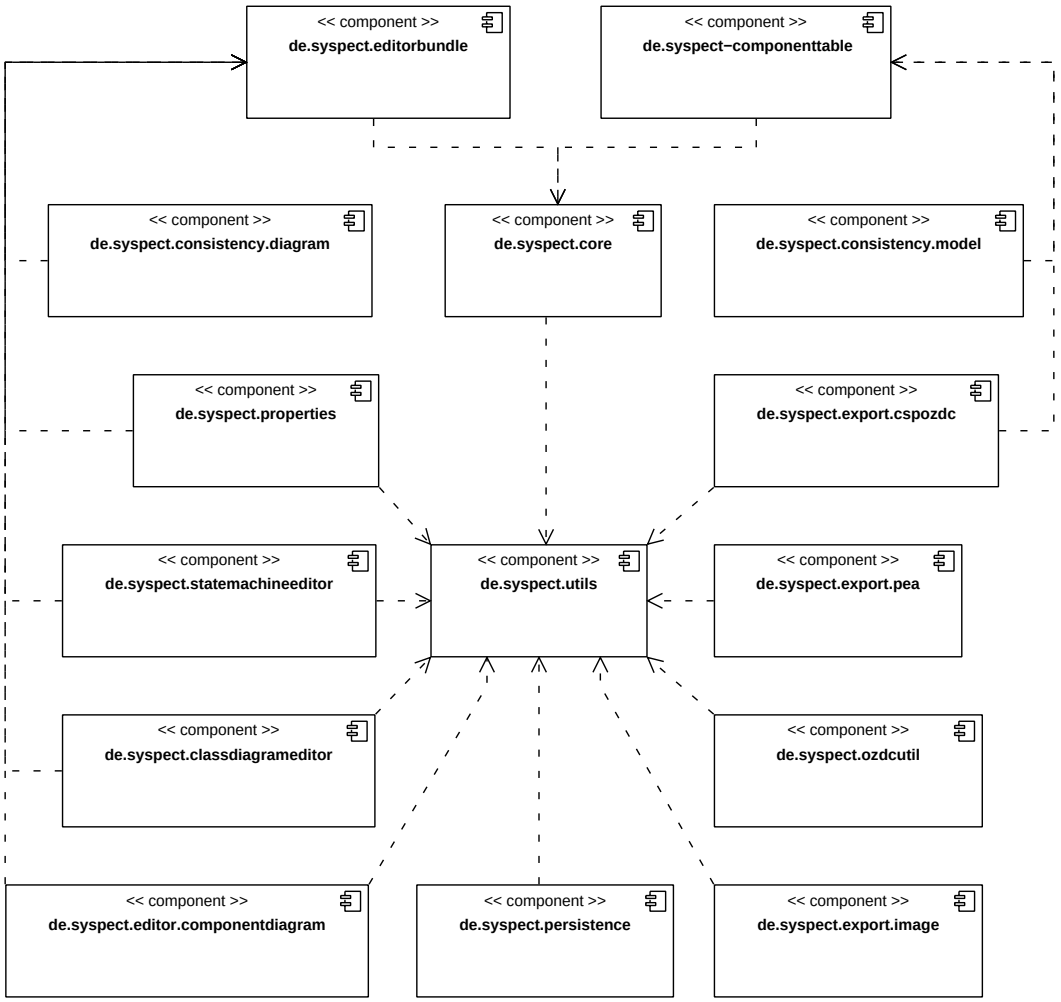


Abbildung 15.3: Abhängigkeiten von Library-Plug-ins

Aktive Plug-ins Auf der anderen Seite gibt es aktive Plug-ins. Diese implementieren *Extension Points*, um sich in das Programm einzugliedern. Es gibt hierbei zwei Arten von *Extension Points*: zum einen sind da jene, die von uns selbst angelegt wurden, zum anderen gibt es eine Reihe *Extension Points*, die wir durch die Rich Client Platform von *Eclipse* erben. So ist es durch letztere möglich, beispielweise sogenannte *view contributions* zu erstellen, um Menüeinträge zu generieren und ähnliches.
Die aktiven Plug-ins bilden eine baumartige Struktur mit dem Haupt-Plug-in `de.syspect.core` als Wurzel. Die Beschreibung der von *Syspect* definierten Extension

Points folgt im Kapitel Feinentwurf.
Die Abhängigkeiten zwischen den Plug-ins über Extension Points sind in Abbildung 15.4 zu sehen und sollen einen Überblick über die Struktur schaffen.

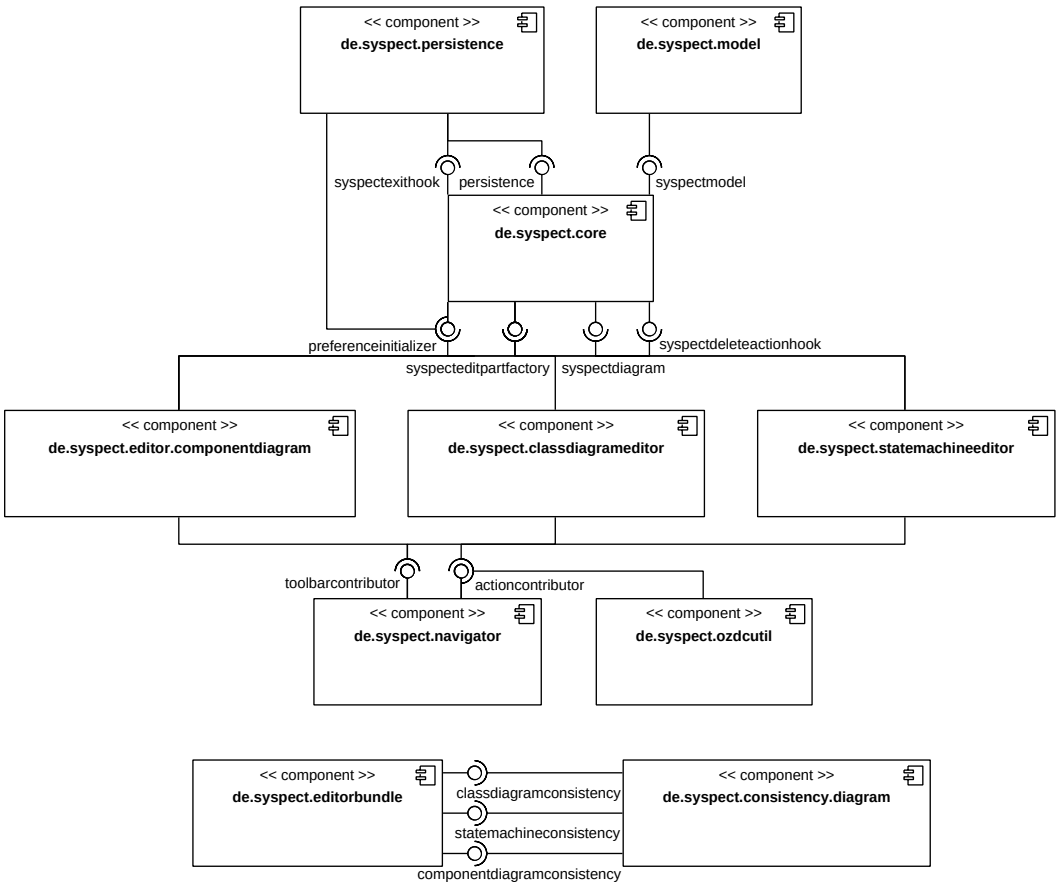


Abbildung 15.4: Extension Points in Syspect

Eclipse-Extension Points

Viele unserer Plug-ins erweitern Extension Points von *Eclipse*. Diese werden hier erläutert, so dass in folgenden Kapiteln, die sich mit den Details der einzelnen Plug-ins beschäftigen, nur darauf verwiesen werden muss.
Das Core-Plug-in erweitert die beiden Extension Points `org.eclipse.help.contexts` und `org.eclipse.help.toc`. Sie ermöglichen das Einbinden eines *Eclipse*-Hilfe-Systems in *Syspect*. Jedes Plug-in, das seine Hilfe-Seiten – in Form von Web-Seiten – bei diesem System registrieren will um Hilfe-Seiten einzubinden, muss ebenfalls den Extension Point `org.eclipse.help.toc` eweitern, der dieses ermöglicht. Auf diese Weise können die Hilfe-Seiten jedes Plug-ins einzeln verwaltet werden und in der *Syspect*-Hilfe sind nur von den Plug-ins Seiten aufrufbar, die im Produkt eingebunden sind. Für eine ausführliche Beschreibung des *Eclipse*-Hilfesystems sei auf die *Eclipse*-Hilfe verwiesen.
Ein weiteres *Eclipse*-Plug-in, das *Syspect* sehr viele Extension Points zur Verfügung

stellt, ist `org.eclipse.ui`. Die folgende Grafik 15.5 soll einen Überblick geben, welche Extension Points von welchen Plug-ins genutzt werden.

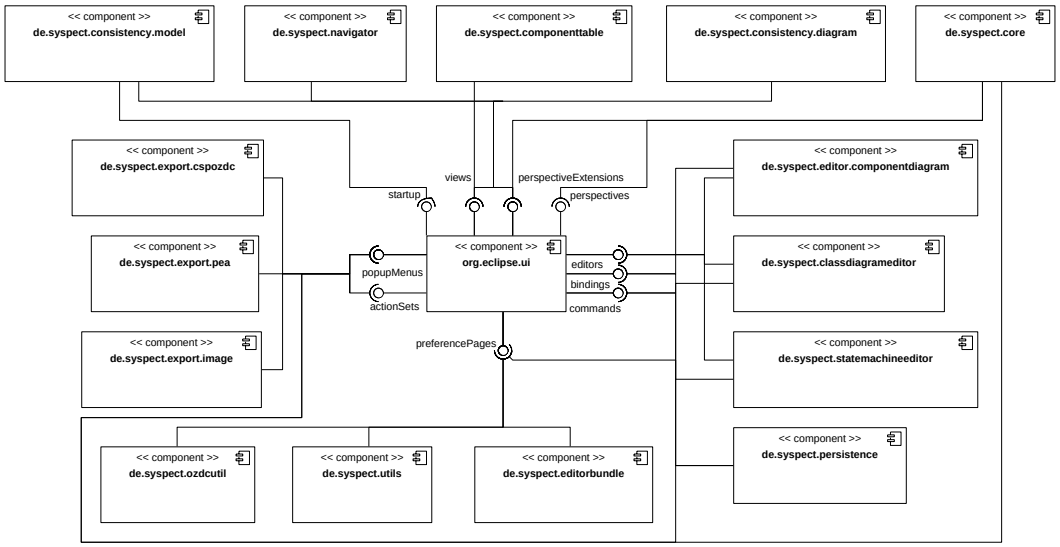


Abbildung 15.5: Extension Points des Plug-ins `org.eclipse.ui`

Das Plug-in `org.eclipse.ui` unterstützt die Interaktion mit und die Erweiterung von der Benutzerschnittstelle (GUI). Wir verwenden dafür folgende Extension Points:

- `org.eclipse.ui.actionSets` wird zum Hinzufügen von Menüs, Menü-Items und Toolbar-Buttons an den dafür vorgesehenen Stellen der Arbeitsfäche verwendet.

- `org.eclipse.ui.commands` wird benutzt, um *Actions* in einem bestimmten *Scope* (Bereich) an ein Command zu binden. Diese Commands werden sowohl verwendet, um Menü-Einträge mit Funktionalität zu belegen, als auch Tastaturkürzel zu ermöglichen. Die Idee von Commands ist hierbei, dass ein Command (z.B. `delete`) in einem Editor anders auszuführen ist, als beispielsweise im Navigator. Dies kann darüber realisiert werden, dass man im Editor-Kontext eine andere Action an das Delete-Command bindet, als beispielweise im Navigator-Kontext. In *Syspect* werden die Commands bislang stehts mit der gleichen Action belegt und sind somit nur dafür da, Tastaturkürzel zu ermöglichen.

  Es sei darauf hingewiesen, dass in diesem Zusammenhang der Begriff „Command“ eine andere Bedeutung hat, als bei den GEF-Editoren (siehe Abschnitt 15.2.1, S. 149 und Abschnitt 15.3.8 und S. 181).

- `org.eclipse.ui.editors` dient zum Hinzufügen von Editoren. Es wird sowohl die Erstellung interner Editoren, die fest integriert sind, als auch die Erstellung externer Editoren, die in einem seperaten Fenster geöffnet werden, unterstützt.

- `org.eclipse.ui.perspectiveExtensions` wird benötigt, damit Plug-ins eine existierende „Perspective“ erweitern können. Eine „Perspective“ definiert den in-

itialen Inhalt von Menü und Toolbar sowie die initialen Views und deren Anordnung innerhalb der Seite.

- `org.eclipse.ui.perspectives` muss erweitert werden um eine eigene „Perspective“ zu definieren.
- `org.eclipse.ui.popupMenus` wird implementiert um zum Kontextmenu weitere Aktionen hinzuzufügen, die zu anderen Plug-ins gehören.
- `org.eclipse.ui.preferencePages` dient zum Hinzufügen von *Preference Pages* von Plug-ins in den von *Eclipse* vorgesehenen Bereich.

  Die *Preferences* sind ein allgemeines Konzept in Eclipse, welches einen Dialog für Einstellungen und Konfigurationen bietet. Es wird ausführlicher in Abschnitt 15.3.2, S. 162 erläutert.
- `org.eclipse.ui.startup` Ein Plug-in, das diesen Extension Point erweitert, registriert sich damit und ist aktiviert wenn das Programm gestartet wird.
- `org.eclipse.ui.views` wird zum Definieren zusätzlicher Views, also visueller Komponenten, innerhalb der Rich Client Application verwendet.

15.3 Feinentwurf

Dieser Abschnitt stellt die einzelnen Plug-ins von *Syspect* vor. Die Funktionalität jedes *Syspect*-Plug-ins wird erklärt und die Schnittstellen zwischen ihnen werden genauer erläutert. Gibt es besondere Konzepte, die bei der Implementierung der einzelnen Komponenten umgesetzt wurden, werden diese zugehörig zu den Plug-ins, in denen sie umgesetzt wurden, erläutert.
Im letzten Abschnitt dieses Kapitels wird auf die Quellcode-Richtlinien eingegangen, die vor der Implementierungsphase festgelegt wurden, um die Lesbarkeit und Einheitlichkeit des Quellcodes zu erhöhen.

15.3.1 Gliederung der Plug-ins

Das Stammpaket jedes Plug-ins trägt die Bezeichnung desselben mit vorangestelltem `de.syspect`. Beispielsweise trägt das Stammpaket des Plug-ins Persistence den Namen `de.syspect.persistence`.
Plug-ins mit Model-View-Controller-Anteil liegt eine einheitliche Paketstruktur zugrunde. Das Stammpaket selbst ist klar nach dem MVC-Pattern strukturiert und enthält, wenn erforderlich, die Pakete `model`, `view` und `controller`. Einzelne Klassen können auch direkt in diesem Paket liegen. Ein einzelnes Paket sollte nicht mehr als 30 Klassen beinhalten. Bei größeren Plug-ins kann daher das Stammpaket weitere Unterpakete enthalten. Diese unterteilen sich dann erneut in die Pakete `model`, `view` und `controller`.
Plug-ins, welche die oben erwähnte Struktur nicht besitzen, sind logisch nach ihrem Inhalt strukturiert. Dies sind unter anderem die Plug-ins `SyspectModel` und `SyspectCSPOZDCExport`.

In allen Plug-ins gibt es außerdem ein `lang`-Paket („language“), welches direkt im Stammpaket oder im `controller`-Bereich jedes Plug-ins liegt. Hierin wurden in der GUI angezeigte Strings in sogenannte Property-Dateien ausgelagert, um eine bessere Handhabung der Textanzeige in der GUI zu bekommen und um Text in mehreren Sprachen zur Verfügung zu stellen.
Die Externalisierung der Strings wurde mit Hilfe von *Eclipse Message Bundles* realisiert. Dieses Prinzip ist auf der Eclipse Homepage [EFIa] beschrieben.
Somit beinhalten die `lang`-Pakete der Plug-ins sowohl die lokalisierten Strings als auch die Methoden, um jene auszulesen.

15.3.2 Syspect Core Plug-in

Das *Syspect Core* ist das Herzstück unserer Applikation. Hier sind zentrale Klassen wie `SyspectApplication` und `SyspectPlugin` angesiedelt, die die Basis unserer Produktes bilden. In diesem Abschnitt werden sowohl die bereitgestellten und benötigten Schnittstellen des Plug-ins erläutert, als auch weitere wichtige Klassen wie der `UndoManager`, der `NameAdministrator` und `SyspectUtil`, das zentrale Prinzip der Eclipse-*Actions* und schließlich das *Syspect*-eigene Prinzip der `DeleteActionHooks`.

Schnittstellen

Syspect Core stellt einige Extension Points zur Verfügung, die im Folgenden beschrieben werden:

- `persistence` Ein Plug-in, welches die Persistenz von *Syspect* übernehmen soll, muss das Interface `IPersistenceManager` implementieren.

- `preferenceinitializer` Der *Initializer* sorgt dafür, dass die Standartwerte der *Preferences* gesetzt und benutzt werden.

  Da von *Eclipse* nur genau ein `PreferenceInitializer` aufgerufen wird, haben wir einen eigenen Extension-Point hierfür eingeführt, der die verschiedenen *Initializer* der Plug-ins einbindet. Er erwartet eine Klasse vom Typ `AbstractPreferenceInitializer`.

- `syspectdeleteactionhook` Diese Schnittstelle wird benötigt, um Editoren benachrichtigen zu können, dass Elemente gelöscht werden sollen. Konkret heißt das, dass wenn ein Element von Typ, der unter `modelclass` angegeben ist, gelöscht werden soll, wird der *delete action hook* aufgerufen, der unter `class` definiert ist (siehe auch Abschnitt 15.3.2, S. 162).

- `syspectdiagram` Dies ist ein Extension Point, der die Erweiterbarkeit von *Syspect* gewährleisten soll. Er muss angesprochen werden, wenn weitere Diagrammeditoren – außer den drei bereits in *Syspect* integrierten – eingebunden werden sollen.

- `syspecteditpartfactory` Jeder Diagrammeditor, der in *Syspect* integriert werden soll, muss eine Klasse haben, die das Interface `org.eclipse.gef.EditPartFactory` implementiert. Desweiteren ist ein Interface anzugeben, dass sich von `IViewableDiagram` ableitet, welche die Diagrammart spezifiziert, für welche der Editor gedacht ist.

- `syspectexithook` Über diesen Extension Point ist es möglich, Operationen zu definieren, die direkt vor Beenden der Applikation durchgeführt werden sollen. Dazu muss das Interface `de.syspect.controller.eclipse.actions.ExitHookAction` implementiert werden.

- `syspectmodel` Um Implementierungen passend zu den Interfaces im Paket `modelinterface` erstellen zu können, werden drei Fabriken benötigt und zwar eine für jede der drei Modelebenen (siehe Abschnitt 15.3.6, S. 168). Da `IProjectManager` (siehe Abschnitt 15.3.6, S. 171) keine dieser drei Ebenen zuzuordnen ist, muss seine Implementierung extra zur Verfügung gestellt werden. Dies sind die Anforderungen an ein Plug-in, das die Modelimplementierung übernehmen soll.

  Die genaue Struktur der Modelschnittstelle ist in das Kapitel über das SyspectModel-Plug-in ausgelagert (siehe Abschnitt 15.3.6, S. 167).

Das Core-Plug-in erweitert viele *Eclipse*-Extension Points. Abbildung 15.6 soll einen Überblick darüber geben.

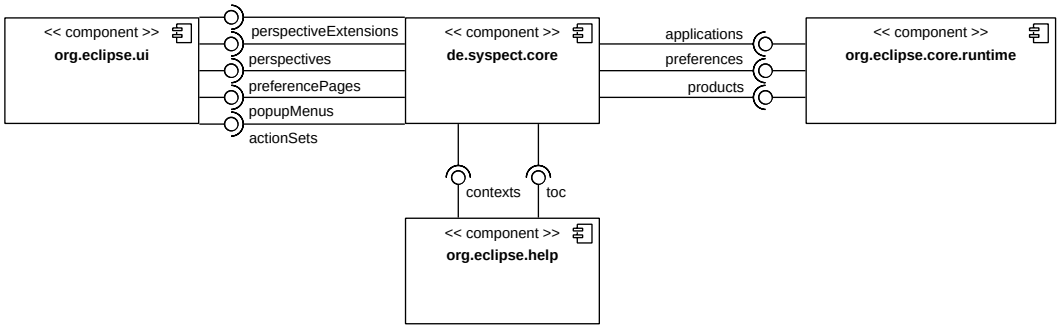


Abbildung 15.6: Von Syspect Core erweiterte Extension Points

Auf die Extension Points `org.eclipse.core.runtime.applications`, `org.eclipse.core.runtime.preferences` und `org.eclipse.core.runtime.products` wird im folgenden Abschnitt Rich Client Platform genauer eingegangen.

Die Plug-ins `org.eclipse.help.contexts`, `org.eclipse.help.toc`, `org.eclipse.ui.actionSets`, `org.eclipse.ui.perspectiveExtensions`, `org.eclipse.ui.perspectives`, `org.eclipse.ui.popupMenus` und `org.eclipse.ui.preferencePages` wurden bereits vorgestellt (siehe Abschnitt 15.2.2, S. 154).

Es gibt weitere Abhängigkeiten zu folgenden Plug-ins:
`de.syspect.utils`, `org.apache.lucene`, `org.eclipse.help.appserver`, `org.eclipse.help.base`, `org.eclipse.help.ui`, `org.eclipse.help.webapp`, `org.eclipse.gef`, `org.eclipse.swt`, `org.eclipse.tomcat`, `org.eclipse.ui.forms` und `org.eclipse.ui.views`.

Rich Client Platform

Syspect setzt auf der *Eclipse Rich Client Platform* auf. Die minimale Menge an Plug-ins, die benötigt wird, um eine *Rich Client Application* zu erstellen, nennt sich *Rich Client Platform*. Für die Funktionalität der Anwendung können natürlich weitere Plug-ins verwendet werden; die Plattform ist nur das Minimum, das zum Erstellen einer Anwendung ausreicht.
Um so ein *Stand-alone*-Programm mit einer grafischen Oberfläche mit *Eclipse* zu erstellen, werden nur zwei Plug-ins (und deren Voraussetzungen) benötigt. Die beiden Plug-ins sind `org.eclipse.ui` und `org.eclipse.core.runtime`. Auf `org.eclipse.ui` wurde bereits im Abschnitt 15.2.2 eingegangen. Dieses Plug-in wird für die grafische Benutzeroberfläche, die Interaktion mit dieser und ihre Erweiterung benötigt. Das Plug-in `org.eclipse.core.runtime` ist das Fundament der Plattform.

Dieses Plug-in wird zum Starten und Laufen eines Programms, sowie für die Verwaltung der Plug-ins benötigt.
Damit sich Syspect Core als Anwendungs-Plug-in gegenüber der *Eclipse*-Plattform registrieren kann, muss es den Extension Point `org.eclipse.core.runtime.applications` erweitern. Dies ist der zentrale Punkt, der es dem Core-Plug-in ermöglicht, die RCP-Anwendung zu konfigurieren.
Syspect Core erweitert noch zwei weitere Extension Points des Runtime-Plug-ins. Die Erweiterung von `org.eclipse.core.runtime.products` ermöglicht es, aus der Anwendung ein Produkt zu machen. Dieses beinhaltet die Anwendung inklusive aller Plug-ins, von denen es abhängt. Zusätzlich können Icons, Namen und About-Infos für das Produkt hinterlegt werden. Das Produkt *Syspect* kann auf Grund dieser Erweiterung außerhalb von Eclipse gestartet werden. Syspect Core erweitert den Extension Point `org.eclipse.core.runtime.preferences` um Default-Werte für die „Preferences“ festzulegen. Das Einstellen dieser kann dann über Preference Pages erfolgen. Diese können hinzugefügt werden, indem `org.eclipse.ui.preferencePages` erweitert wird.

UndoManager

Der `UndoManager` verwaltet sämtliche Aktionen des Benutzers, die *rückgängig* zu machen bzw. *wiederherstellbar* sind. Solche sind in Klassen gekapselt, die das Interface *IUndoable* implementieren.
Die Größe der *History*, also die Anzahl der Aktionen, die hintereinander wieder rückgängig gemacht werden können, ist über die *Preferences* einstellbar.

NameAdministrator

Der `NameAdministrator` sorgt für die Eindeutigkeit von Namen von Modellelementen innerhalb eines Projektes. D.h. bei der Erzeugung oder Umbenennung von Modellelementen muss der neue Name hier registriert und ggf. der alte Name ausgetragen werden. Ebenso stellt diese Klasse garantiert eindeutige *default*-Namen zur Verfügung, damit der Benutzer nicht gezwungen ist jedem neuen Element sofort einen eigenen Namen zu geben.
Da die hier gespeicherte Abbildung von Elementen auf Namen nicht Teil der persistenten Datenhaltung ist, wird es beim Laden eines Projektes neu erstellt.

SyspectUtil

Eine besondere Klasse im Core-Paket ist die Klasse `SyspectUtil`. Hier sind allgemeine Hilfsmethoden angesiedelt, die z.B. Programmlogik für das Modell beinhalten. Dies hat die Vorteile, dass Datenhaltung und Logik getrennt sind und häufig verwendete Methoden an einem zentralen Punkt von mehreren Komponenten gemeinsam benutzt werden können.
Es ist aber zu beachten, dass die hier definierten Methoden das Modell nicht verändern. Dies darf in unserer Architektur nur über `Commands` geschehen (siehe 15.3.8, S. 181).

Actions

Das Konzept der `Actions` ist ein von Eclipse bereitgestellter Mechanismus, bei dem durch Auswahl, z.B. eines Menüpunktes, bestimmte Aktionen ausgeführt werden. Zu diesem Zweck muss die abstrakte Klasse *Action* des Paketes `org.eclipse.jface.action` erweitert werden. Diese stellt eine Standardimplementierung für das Erstellen von konkreten `Actions` zur Verfügung. Insbesondere muss die *run*-Methode, die das Verhalten der `Action` spezifiziert, implementiert werden. Eine Möglichkeit, diese `Actions` in eine GUI von Eclipse bzw. der `RCP` einzubinden, ist über entsprechende Extension Points. Diese werden in der plugin.xml definiert. U.a. gibt es die Extension Points `editorActions`, `viewActions` oder `popupMenus`. Abbildung 15.7 zeigt eine solche Definition. In diesem Beispiel wird eine `Action` für ein Kontextmenü in einer `View` definiert. Über Attribute und Tags werden entsprechende Einstellungen für eine `Action` durchgeführt. Für genauere Informationen sei auf die Eclipse-Dokumentation verwiesen.

```
<extension point="org.eclipse.ui.popupMenus">
     <viewerContribution
      id="de.syspect.navigator.view.delete"
      targetID="de.syspect.navigator.view.diagram">
          <action
           class="de.syspect.controller.diagram.actions.DeleteAction"
           icon="images/delete.gif"
           id="de.syspect.navigator.view.delete.entry"
           label="Delete"
           menubarPath="new.menu">
               <enablement>
                    <objectClass
                     name="de.syspect.modelinterface.IDeletable"/>
                </enablement>
          </action>
     </viewerContribution>
</extension>
```

Abbildung 15.7: Bereitstellen von `Actions` über Extension Point

Die `Action`-Klassen müssen entsprechend das Interface *IActionDelegate* aus dem Paket `org.eclipse.ui` implementieren (vgl. Abbildung 15.8). Genauer gesagt muss eine `Action`, die für das Menü bzw. die Werkzeugleiste der Workbench bereitgestellt werden soll, das *IWorkbenchWindowActionDelegate* Interface implementieren. Für das Hinzufügen von `Actions` zu z.B. Werkzeugleisten oder Kontextmenüs in `Views` muss die `Action`-Klasse das *IViewActionDelegate* Interface implementiert werden. Entsprechend werden `Actions` für Editoren bereitgestellt, wenn sie das *IEditorActionDelegate* Interface implementieren. Speziell können `Actions` für bestimmte Objekt-Typen definiert werden. Hierzu muss das *IObjectActionDelegate* Interface implementiert werden.

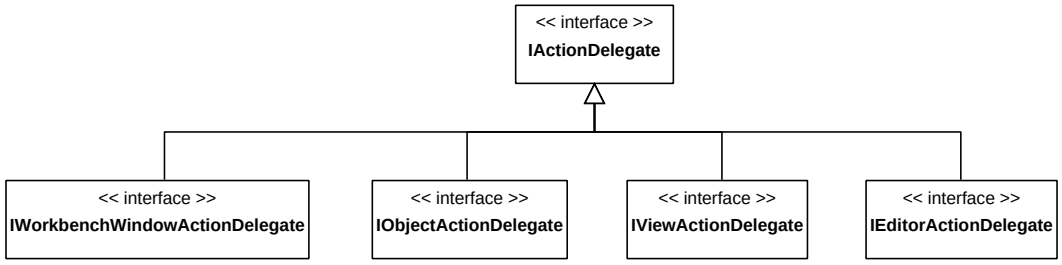


Abbildung 15.8: *IActionDelegates*-Struktur

DeleteActionHooks

Das Konzept der `DeleteActionHook`s wurde entwickelt, um Elemente aus der `Model`- und der `Viewable`-Ebene zu löschen. Beim Löschen eines `Model`-Elements, müssen

1. auch alle mit diesem Element in Verbindung stehenden `Model`-Elemente,
2. alle dem Element zugehörigen `Viewable`-Elemente gelöscht werden.

Fall 2 muss von jedem Plug-in, welches `Viewable`-Elemente erzeugt, behandelt werden, da nur in diesen Plug-ins genügend Informationen über die Abhängigkeiten der `Viewable`-Elemente vorliegen. Das Löschen der `Model`-Elemente, die mit dem zu löschenden Objekt in Verbindung stehen, wird im Core Plug-in durchgeführt, da ansonsten verschiedene Implementierungen z.B. in den Diagrammeditoren zu inkonsistenten Zuständen eines Projekts führen könnten. So werden zum Beispiel beim Löschen einer `ICapsule` alle mit der Klasse verbundenen `IClassConnection`s gelöscht, das Löschen der `IVCapsule` und der `IVClassConnection`s wird vom Classdiagrameditor Plug-in übernommen (für eine nähere Erläuterung der *Model*-Elemente siehe Abschnitt 15.3.6, S. 167).
Die `DeleteActionHook`s rufen zum Löschen der Elemente Commands (siehe Abschnitt 15.3.8, S. 181) auf, jedoch werden diese Commands nicht an den `UndoManager` gemeldet, da ein Löschvorgang als atomare Aktion behandelt wird, um inkonsistenten Zuständen des Projekts vorzubeugen. Um `Model`-Elemente zu Löschen, wird das `DeleteCommand<T>` aus dem Core Plug-in erzeugt, so dass für jedes gelöschte Element wieder dessen `DeleteActionHook`s aufgerufen werden. Um `Viewable`-Elemente zu entfernen, wird das `DeleteCommand<T,S>` aus dem *SyspectDiagramEditorBundle* (siehe Abschnitt 15.3.8, S. 180) benutzt, welches ein `Viewable`-Element aus der Liste der Kinder seines Eltern-Objektes löscht.

Preferences

`PreferencePages` sind ein Konzept von Eclipse, um einfaches konfigurieren von Programminhalten zu bewerkstelligen. Ähnlich wie bei den Wizards Seiten, sogenannten PreferencePages, welche ueber `FieldEditoren` leicht zu bauen sind. Diese Editoren greifen auf Konstanten zu welche vom schon erwähnten PreferenceInitilizer ihren Defaultwert erhalten.

Beim bauen der Seite ist zu beachten das diese von FieldEditorPreferencePage abgeleitet ist, um an die FieldEditoren heranzukommen, und IWorkbenchPreferencePage implementiert. Beispiele für FieldEditoren sind DirectoryFieldEditoren, zum auswählen von Verzeichnissen, und RadioGroupFieldEditor für die Auswahl einer Sache aus mehreren möglichen. Die Seite muss in der plugin.xml unter Extensions eingetragen werden, damit sie dort auftaucht wo sie hinsoll.
Die Konstante, die mit dem FieldEditor bearbeitet wird, muss in der PreferenceConstants des jeweiligen Plug-ins eingetragen sein, und mit dem PreferenceInitializer initialisiert werden.

15.3.3 SyspectNavigator Plug-in

Bei diesem Plug-in handelt es sich um eine Ergänzung der Benutzeroberfläche. Es stellt Baumansichten des *Models* zur Verfügung (für eine nähere Erläuterung des *Models* siehe Abschnitt 15.3.6, S. 167). Es bietet drei verschiedene Darstellungen:

- *Diagram Explorer*: In dieser Darstellung werden in einem Projekt nur die Diagramme dargestellt. Dadurch ergibt sich eine Ansicht auf die *Viewable*-Ebene des *Models*.
- *Model Explorer*: Hierbei werden in einem Projekt nur die Elemente der *Model*-Ebene dargestellt. Es werden also keine Diagramme als Kinder von Projekten angezeigt. Zustandsautomaten tauchen allerdings in dieser Ansicht als Kinder ihrer *Capsules* auf.
- *Project Explorer*: In dieser Ansicht werden sowohl die *Viewable*-Ebene als auch die *Model*-Ebene angezeigt.

Schnittstellen

Das Navigator Plug-in stellt die Extension Points `actioncontributor` und `toolbarcontributor` zur Verfügung.
Der `toolbarcontributor` Extension Point ermöglicht es anderen Plug-ins (zum Beispiel Editoren) Actions in der Werkzeugleiste des Navigators unterzubringen.
Der `actioncontributor` Extension Point bietet eine Schnittstelle, um anderen Plug-ins zu ermöglichen, auf „Doppelklicken“ in der Baumansicht zu reagieren. Hierzu muss nur das Interface `de.syspect.navigator.ep.IDoubleClickContributor` implementiert werden.

Das Plug-in erweitert die *Eclipse*-Extension Points
`org.eclipse.help.toc`,
`org.eclipse.ui.perspectiveExtensions` und
`org.eclipse.ui.views`
(siehe Abschnitt 15.2.2, S. 154).

Es gibt weitere Abhängigkeiten zu folgenden Plug-ins:
`de.syspect.core`,

`org.eclipse.core.runtime`,
`org.eclipse.gef` und
`org.eclipse.ui.views.properties.tabbed`.

Drag and Drop

Das Navigator Plug-In ist eine `DragSource`. Das heißt, dass per `Drag and Drop` Elemente aus dem Navigator in andere Ansichten gezogen werden können. Insbesondere wird diese Funktionalität genutzt, um bereits existierende Modelelemente zu Editoren hinzuzufügen.
Hierzu wird die von Eclipse bereitgestellt Funktionalität (`org.eclipse.swt.dnd`) genutzt. Als Daten werden vom Navigator ausschließlich Namen (als `Strings` von `Model`-Elementen) übergeben. Zieht der Benutzer ein Element der `Viewable`-Ebene mit der Maus, so wird, falls das Element ein Model hat, der Namen des entsprechenden *Model*-Elements genutzt.

15.3.4 SyspectPersistence Plug-in

Das Persistence Plug-in ist für das Laden und Speichern des *Models* zuständig. Diese Funktionalität wird unter Benutzung des `XMLEncoders` (`java.beans.XMLEncoder`) bereitgestellt. Da dafür die Kenntnis der Klassenstruktur des *Models* notwendig ist, arbeitet dieses Plug-in nicht auf der Interfacestruktur. Daher ist es nur in der Lage, Daten in Form von Instanzen der Klassen im *SyspectModel*-Plug-in zu speichern beziehungsweise zu laden.
Zusätzlich zu der Grundfunktionalität stellt das Plug-in auch `Wizards` beziehungsweise `Actions` zum Speichern, Laden und neue Projekte Erstellen bereit.

Schnittstellen

Das Persistence-Plug-in erweitert drei Extension Points des Syspect Core Plug-ins:
`de.syspect.core.persistence`,
`de.syspect.core.preferenceinitializer` und
`de.syspect.core.syspectexithook` (Beschreibung siehe Abschnitt 15.3.2, S. 158).

Zusätzlich erweitert es folgende *Eclipse*-Extension Points:
`org.eclipse.help.toc`,
`org.eclipse.ui.actionSets`,
`org.eclipse.ui.bindings`,
`org.eclipse.ui.commands`,
`org.eclipse.ui.popupMenus` und
`org.eclipse.ui.preferencePages` (siehe Abschnitt 15.2.2, S. 154).

Das Syspect Persistence Plug-in besitzt weitere Abhängigkeiten zu folgenden Plug-ins:
`de.syspect.model`,
`de.syspect.utils`,
`org.eclipse.core.runtime`,

`org.eclipse.draw2d` und
`org.eclipse.ui.views`

Die Klasse `de.syspect.persistence.PersistenceManager` ist die Schnittstelle des Plugins nach außen. Sie bietet sowohl Methoden zum Laden als auch zum Speichern eines Projektes.

Dateiformat

Im Folgenden wird beschrieben, in welchem Format Projektdaten gespeichert, transportiert und geladen werden können.
Die Daten werden in einer Datei mit der Endung `.zsp` gespeichert. Hierbei handelt es sich um ein Zip-Archiv. Dadurch ist es für eine spätere Erweiterung von *Syspect* einfach, eigene Daten abzuspeichern. Es kann einfach eine weitere Datei in das Archiv eingefügt werden.
In dem Archiv liegt eine Datei, die die tatsächlichen Daten enthält. Diese Datei heißt `view.xml`. In dieser wird ein Projekt als XML gespeichert. Hierzu wird der in Java eingebaute `XMLEncoder` verwendet. Dieser erzeugt eine textuelle Darstellung einer Java-Bean. Zum Speichern des *Models* werden für mehrere Klassen spezielle `Persistence-Delegates` verwendet:

- `Rectangle`: Da die Eclipse-Implementation von Rectangle im *Model* verwendet wird, wurde ein `Persistence-Delegate` definiert, der den Konstruktor Rectangle(int, int, int, int) benutzt.
- `AbsoluteBendpoint`: Ein `Persistence-Delegate` zur Benutzung des Konstruktors AbsoluteBendpoint(int, int) wurde erstellt.
- `Point`: Es wird der Konstruktor Point(int, int) verwendet.
- *Enums*: Für die Klassen `EVisibility`, `IConnectionLabel.EPosition`, `EMethodType`, `EProblemStatus` und `EZStatus` wurde ein gemeinsamer `Persistence-Delegate` implementiert, welcher die Methode `valueOf(String)` verwendet.

Zudem wurden die Attribute `fileName`, `path` und `dirty` aus `ProjectImpl`, sowie `label` aus `ConnectionLabelImpl` auf `transient` gesetzt, da diese zur Laufzeit generiert werden. Das Attribut `parent` aus `ProjectImpl` wurde ebenfalls auf `transient` gesetzt, damit nicht alle offenen Projekte gespeichert werden.

Ein leeres Projekt, das nur die System-Definitionen enthält, sieht dementsprechend folgendermaßen als XML aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_07" class="java.beans.XMLDecoder">
<object id="ProjectImpl0" class="de.syspect.model.ProjectImpl">
  <void id="HashSet0" property="children">
```

```
   <void method="clear"/>
   <void method="add">
    <object class="de.syspect.model.SystemDefinitionImpl">
     <void property="parent">
      <object idref="ProjectImpl0"/>
     </void>
    </object>
   </void>
  </void>
  <void property="children">
   <object idref="HashSet0"/>
  </void>
  <void property="name">
   <string>ProjectName</string>
  </void>
 </object>
</java>
```

Wie zu erkennen ist, werden die als `transient` markierten Attribute nicht ausgegeben.

15.3.5 SyspectUtils Plug-in

Das Plug-in stellt Basisfunktionalitäten zur Verfügung, die nicht von anderen Plug-ins abhängen. Beispielsweise beinhaltet die Klasse `LayoutsAndDatas` aus dem Paket `de.syspect.utils.view` statische Funktionen, die standardisierte Layouts für Komponenten der Benutzeroberfläche erzeugen. Des Weiteren werden allgemeine `Widgets` und verschiedene `Wizards` für das Syspect-Projekt implementiert. Die `Wizards` erweitern die abstrakte Klasse `AbstractSyspectWizard` (vgl. Abbildung 15.9), die eine Initialisierungsmethode bereitstellt.

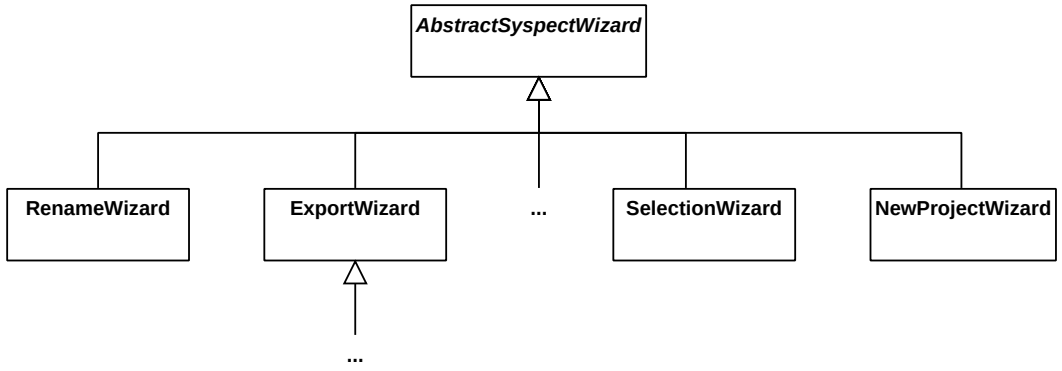


Abbildung 15.9: Die Verberbungshierarchie von `AbstractSyspectWizard`

Dieses Plug-in erleichtert die Erweiterbarkeit von *Syspect* insofern, als dass von mehreren Plug-ins gemeinsam genutzte Funktionalität vorläufig an dieser Stelle gesammelt

werden kann. Natürlich muss es in regelmäßigen Abständen Reengineering-Phasen geben, in denen untersucht wird, inwieweit Funktionalität thematisch zusammenpassend in ein separates Plug-in – in unserem Falle z.B. eine *Wizard*-Bibliothek – ausgelagert werden kann. Ist dies aber der Fall, ist ein solches *utils*-Plug-in ein wichtiges Hilfsmittel, um die Flexibilität während der Entwurfs- und Implementierungsiterationen zu erhöhen.

Schnittstellen

Das Plug-in erweitert den Extension Point `org.eclipse.help.toc`. Weitere Abhängigkeiten hat SyspectUtils nur von den *Eclipse* Plug-ins `org.eclipse.core.runtime` und `org.eclipse.ui` (siehe Abschnitt 15.2.2, S. 154).

15.3.6 SyspectModel Plug-in

Dieses Plug-in enthält die Implementierung für das Model und stellt somit die Struktur für die Datenhaltung von *Syspect*.

Schnittstellen

Dieses Plug-in dockt an dem Extension Point `de.syspect.core.syspectmodel` vom SyspectCore (siehe Abschnitt 15.3.2, S. 158) und dem *Eclipse* Extension Point `org.eclipse.help.toc` (siehe Abschnitt 15.2.2, S. 154) an.

SyspectModel besitzt folgende weitere Abhängigkeiten:

- `org.eclipse.core.runtime.Plugin`, da die Eclipse-Architektur fordert, dass ein Plug-in eine Klasse haben muss, die das Interface `BundleActivator` implementiert.
- `org.eclipse.draw2d`, weil Klassen wie `Point` und `Rectangle` verwendet werden, um grafische Informationen zu speichern.
- Aus `org.eclipse.ui.views` werden Elemente wie `IPropertySource` für das Event-Konzept verwendet (siehe Abschnitt 15.3.6, S. 169).

Darüber hinaus werden in dem Enum `EPropertyConstants` dem Java-Bean-Konzept folgend, die Namen der Attribute der Modelklassen aufgelistet. Dies wird benötigt, damit die *Tabbed Property View* (siehe Abschnitt 15.3.12, S. 192) darauf zugreifen kann.
Im folgenden werden nun zunächst einige Konzepte vorgestellt, die durchgehend im gesamten Model verwendet werden. Anschließend werden thematisch sortiert in einem ausgewählten Kontext alle Modelelemente vorgestellt.

Übergeordnete Konzepte

- **Namenskonventionen**

  - Alle Namen in der Interface-Struktur im *Model* haben ein „I“ am Anfang des Namens.
  - Ein Interface der *Viewable*-Ebene hat ein „V“ nach dem „I“ im Namen, wenn es ein dazu korrespondierendes Interface (dann ohne „V“) auf der *Model-Ebene* gibt (Informationen zu den Modelebenen siehe unten).
  - Eine Klasse, die ein *Model*-Interface implementiert, trägt denselben Namen wie dieses Interface, nur ohne „I“ zu Beginn und mit „Impl“ am Ende.
  - Verwendete Enums beginnen mit einem „E“.

- **Eltern-Kind-Beziehungen** Die Datenhaltung von *Syspect* basiert auf einer Baumstruktur, d.h. die meisten Elemente sind über Eltern-Kind-Beziehungen miteinander verknüpft. Klassen, die Elternelement sind, müssen das Interface `IParent` implementieren und Klassen die Kindelement sind, müssen das Interface *IChild* implementieren.

  Abbildung 15.10 zeigt eine Übersicht dieser Struktur. Vererbungsbeziehungen sind dann mit eingezeichnet, wenn dadurch das Verständnis erleichtert wird – eine vollständige Darstellung der Vererbungsbeziehungen ist in den nachfolgenden Abschnitten enthalten.

  Bei der Modellierung der Eltern-Kind-Beziehung wurden folgende Anforderungen berücksichtigt:

  - Die Beziehungen sollten gegenseitig aufeinander abgestimmt sein, d.h. genau dann wenn ein Element A Vater von einem Element B ist, soll B Kind von A sein.
  - Die Beziehungen sollen sich daran orientieren, was im Navigationsbaum angezeigt werden soll, damit dieser möglichst einfach zu implementieren ist.

  Zu beachten ist jedoch, dass nicht alle Klassen des Modells über die Eltern-Kind-Beziehungen zu erreichen sind. Das sind

  - alle Verbindungen (*Connections*),
  - alle Elemente der View-Ebene,
  - alle Z-Elemente
  - und die DC-Elemente.

- **Die drei Ebenen im Syspect-Model** Es gibt drei Ebenen, denen fast alle Modellelemente zugeteilt werden können:

  - Die „reine“ *Model-Ebene* beinhaltet alle wesentlichen Elemente des UML-Profils (siehe Kapitel 9, S. 77 und Kapitel 14.2, S. 143) als Interfaces sowie sinnvolle abstrakte Superinterfaces. So gibt es z.B. für das *Capsule*-Element das Interface *ICapsule*.

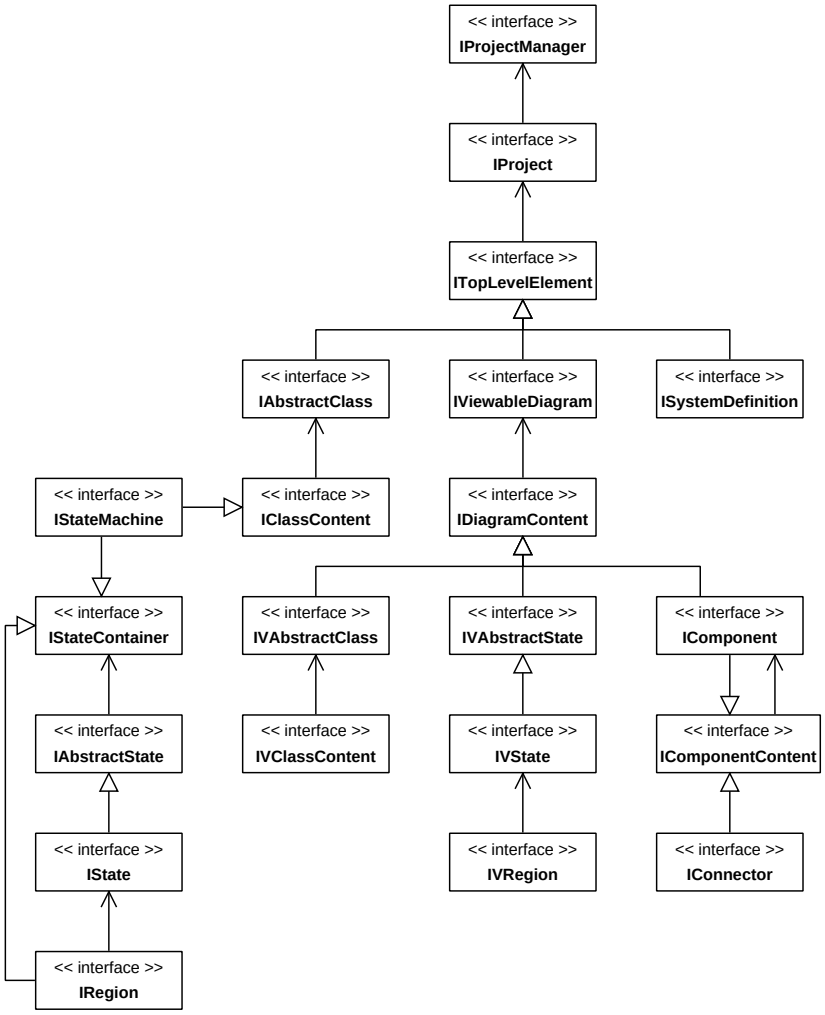


Abbildung 15.10: Eltern-Kind-Beziehungen im *Syspect* Model (in Pfeilrichtung gelesen ist die Semantik „ist Kind von“)

  - Die *Viewable-Ebene* ist eine Art „Ansichts“-Ebene: Es gibt z.B. mehrere Ansichten auf eine *Capsule*, da sie beliebig oft in Klassendiagrammen oder als *Component* in Komponentendiagrammen auftauchen kann. Um dies zu ermöglichen, sind auf dieser Ebene die Interfaces `IVClass` und `IComponent` angesiedelt. In diesen Klassen wird gespeichert, welche Attribute und Methoden (bzw. welche bereitgestellten und benötigten Interfaces) in der jeweiligen Ansicht gezeigt werden sollen.
  - Schließlich geht es in der *View-Ebene* nur um die rein grafischen Elemente. Das heißt, dass – dem Beispiel folgend – sowohl für jedes `IVClass`-Objekt als auch für jedes `IComponent`-Objekt ein `IConstrainedNode`-Objekt existiert, dass speichert, wo es wie groß in einem Diagramm auftaucht.

- **Das Event-System von *Syspect*** Allgemein besteht der Sinn von Events darin, dass Veränderungen, die an einer Stelle im Programm stattfinden, an alle anderen

Teile, die Informationen über diese Veränderung benötigen, weitergeleitet werden.

Beispielsweise ändert ein Benutzer im Klassendiagrameditor den Namen einer Methode. Dazu wird – auf der Modelebene betrachtet – die Methode `setName` in der Klasse `CommentedNamedElement` aufgerufen, die nach Änderung des Names den Event `NAME_CHANGED` feuert.

Möchte eine Klasse über Events in einer Modelklasse informiert werden, kann sie sich über die Methode `addPropertyChangeListener` als Listener für dieses Modelelement eintragen. Außerdem werden Events an den jeweiligen Parent einer Klasse weitergereicht. Ein aufgetretener Event propagiert sich durch den ganzen Baum der Parent/Child-Hierarchie bis nach oben zum *project manager* hin durch.

Im Beispiel der Namensänderung würde das Event `NAME_CHANGED` von der Methode an die Klasse, in der die Methode liegt, weitergeleitet. Die Klasse übergibt dieses Event dann wiederum an ihren Parent, also an das Projekt. Der Grund dafür ist, dass bei Änderung des Namens einer Methode im Klassendiagramm es möglicherweise nicht ausreicht, nur die Methode neu zu zeichnen. Möglicherweise muss die ganze Klasse oder gar das ganze Diagramm neu gezeichnet werden, um auf die Veränderung zu reagieren.

Tritt ein Event in View- oder Viewable-Ebene auf, wird dieses an das dazugehörige Modelelement weitergegeben. Dadurch wird eine gewisse Fehlertoleranz erreicht, da es nicht nötig ist, sich als Listener für einzelne grafische Elemente wie zum Beispiel `ConstrainedNode` einzutragen. Stattdessen erhält ein Modelelement auch alle Informationen über Veränderungen in den dazugehörigen grafischen Elementen.

Alle in *Syspect* verwendeten Events sind in dem Enum `EEventsConstants` definiert.

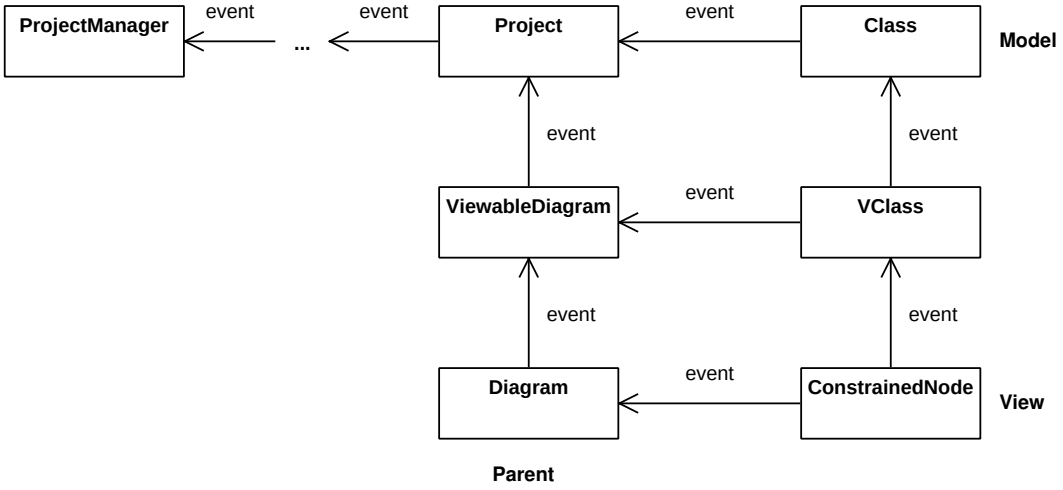


Abbildung 15.11: Ein in einer „ConstrainedNode“-Klasse aufgetretener Event wird an die Modelebene weitergeleitet (vertikale Richtung) und in der Parent/Child-Hierarchie zur Wurzel propagiert (horizontale Richtung).

Die Strategie, wie in *Syspect* Events gefeuert werden, ist somit ein Kompromiss zwischen einer aufwendigen Logik, die möglichst wenig Events nur an die Elemente feuert, die diese auch benötigen und einer simplen Logik, die aber so viele Events feuert, dass die Performanz des Gesamtsystems über Gebühr in Mitleidenschaft gezogen wird.

Kontext: Hilfreiche Superinterfaces

In Abbildung 15.12 sind einige verwendete Superinterfaces und ihre Vererbungszusammenhänge skizziert. Ihre Namen sind zum großen Teil selbsterklärend und funktionieren nach folgendem Prinzip: Alle Klassen, die `INamedElement` implementieren, können benannt werden, alle Klassen die `ICommented` implementieren, können mit einem Kommentar versehen werden usw.

Eine Ausnahme bildet `IPotentiallyProblematicElement`, denn in der gegenwärtigen Implementierung gibt es nur eine Klasse – `ProblemSupport` –, die dieses Interface implementiert. Diese ist per Komposition in alle betreffenden Klassen eingebunden.
Es sei aber darauf hingewiesen, dass es möglich wäre, bei einer anderen Modelimplementierung dies ebenfalls durch Vererbung zu lösen.

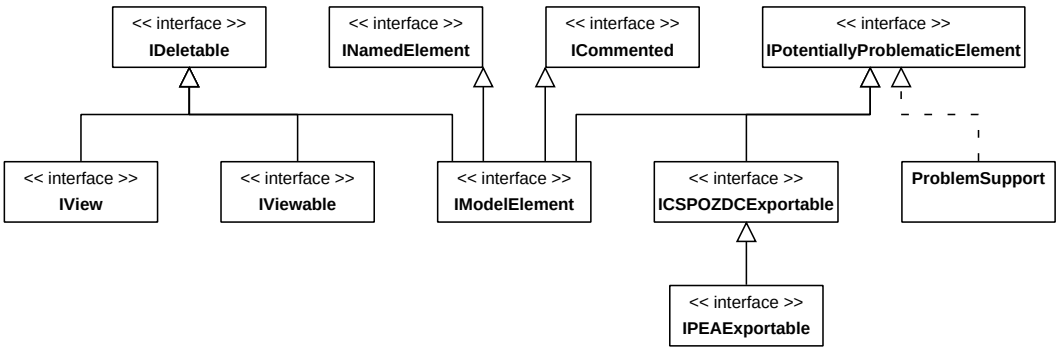


Abbildung 15.12: Superinterfaces

Kontext: Projektmanager und Projekte

Die Schnittstelle, über die sämtliche Modelldaten erreicht werden können, heißt `IProjectManager`. Sie verwaltet eine Liste von `IProject`-Instanzen, welche jeweils alle Modelldaten für ein Projekt beinhalten. Die Klasse `IProject` verwaltet die ihm untergeordneten Modellelemente wie folgt:

- In einer Liste von `IAbstractClass`-Instanzen werden sämtliche Klassen und Interfaces gespeichert.
- In einer Liste von `IViewableDiagram`-Instanzen werden sämtliche Diagramme gespeichert.
- In einer `ISystemDefinition`-Instanz werden globale Z-Definitionen gespeichert.

Zum besseren Zugriff auf diese Modellelemente implementieren sie alle das Interface `ITopLevelElement`, so dass bei einem Aufruf von `getChildren` eines Projektes sämtliche dieser Modellelemente zurückgegeben werden.

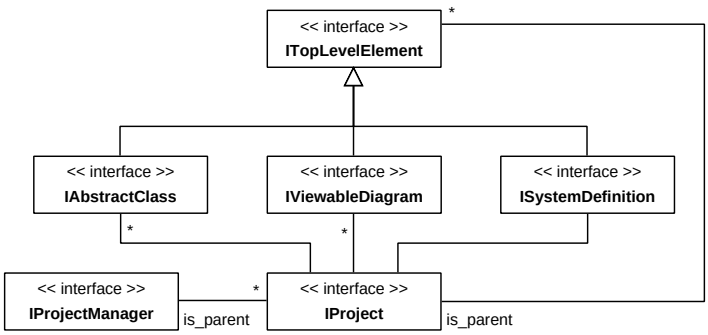


Abbildung 15.13: Projektmanager und Projekte

Kontext: Klassen und Interfaces

Das abstrakte Oberinterface für die Model-Repräsentationen von *Capsule*, *Data* und *Interface* ist `IAbstractClass`. `IAbstractClass`-Instanzen können `IClassConnections` als ausgehende und abgehende Verbindungen haben und sie sind immer Kind ihres Projektes. Sie selbst sind Eltern von `IClassContent`-Instanzen.
Die erbenden Interfaces der `IAbstractClass` sind `IClass` und `IInterface`. Sie unterscheiden sich wie folgt:

- `IClass` hat als `IClassContent` `IAttribute`- und `IMethod`-Objekte. `IClass` ist wie `IAbstractClass` abstrakt und hat als konkrete Unterklassen `ICapsule` und `IData`. Diese beiden unterscheiden sich dadurch voneinander, dass `ICapsule` eine `IStateMachine` (zusätzlich als `ClassContent`) und ein `Set` von `IDCTag`-Objekten besitzt. `IDCTag` kapselt eine `IDCFormula` und implementiert das `INamedElement`-Interface. Die `IStateMachine` und andere zugehörige Klassen werden in dem entsprechenden Abschnitt (Seite 175) behandelt.

- `IInterface`-Instanzen haben als `IClassContent` `IMethodSignature`-Objekte.

Einige Details zum `IClassContent`:

- `IMethodSignature` und `IAttribute` implementieren das `IFeature`-Interface, welches besagt, dass sie eine Sichtbarkeit haben (Wir haben uns für den Namen Feature entschieden, weil in der UML auch Elemente mit einer Sichtbarkeit Feature genannt werden).

- Jede `IMethod` besitzt eine `IMethodSignature`. Wenn z.B. eine Realisierungsbeziehung zwischen einer `ICapsule` und einem `IInterface` erstellt wird, wird für jede `IMethodSignature` in dem Interface jeweils eine `IMethod` erstellt, die die entsprechende `IMethodSignature` als Attribut hat.

- Es gibt einen Ausnahmefall, in dem `IMethodSignature`-Instanzen `IClass`-Instanzen als Elternelemente haben können, und zwar wenn sie Signaturen von lokalen Methoden sind.

Die Beziehungen von `IAbstractClass`-Objekten zu Z-Elementen wird in dem Abschnitt über Z-Elemente (S. 173) behandelt.

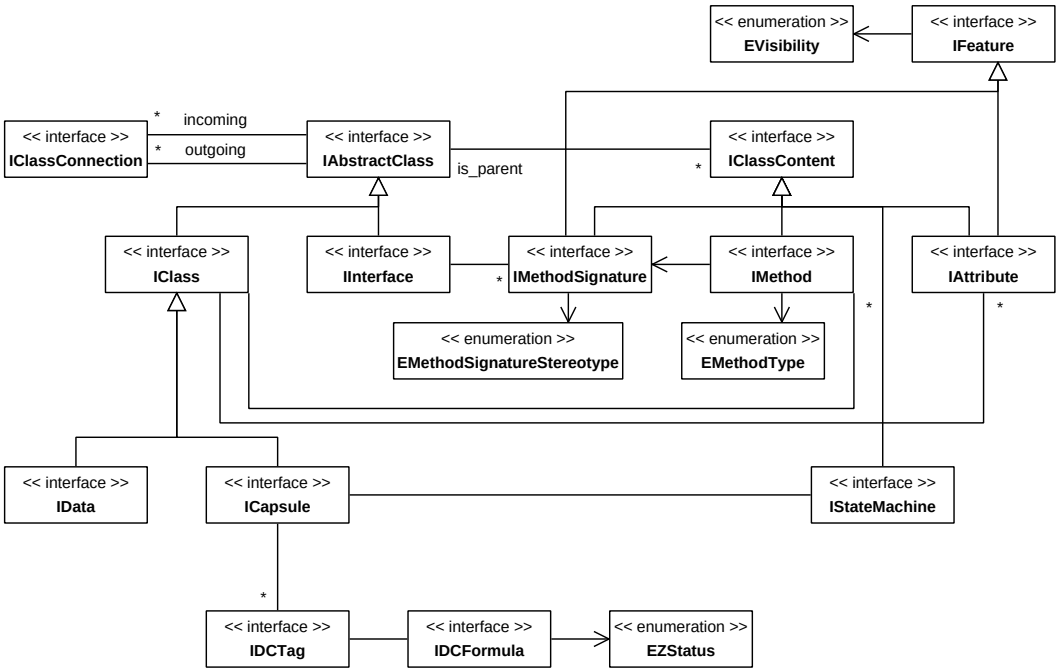


Abbildung 15.14: Klassen und Interfaces

Kontext: Z-Elemente

Alle Z-Elemente leiten sich von `IZElement` ab. Jedes `IZElement` hat einen `EZStatus`. Im folgenden werden die einzelnen Unterklassen von `IZElement` aufgelistet und beschrieben, wo sie vorkommen:

- Ein `IZType` speichert in einem `String` einen Z-Typ. Jedes `IAttribute` und jede `IZDeclaration` haben einen `IZType`. Außerdem leitet sich `IData` von `IZType` ab. Dies hat folgenden Grund: Wenn z.B. eine `ICapsule` durch eine `IAggregation` ein `IData`-Objekt beinhaltet, so bekommt sie dieses als Attribut. Da `IAttribute`-Objekte einen `IZType` brauchen ist `IData` ein `IZType`.

- Die schon erwähnte `IZDeclaration` findet in der `IMethodSignature` ihre Anwendung. Eine `IMethodSignature` hat jeweils mehrere *IN*-, *OUT*- und *SIMPLE*-`IZDeclarations`.

- Die Klasse `IZPredicate` wird zum einen als *ENABLE* und *EFFECT* in der `IMethod` verwendet und zum anderen als *INIT* und *INVARIANT* in `IClass`.

- Sowohl die `ISystemDefinition` als auch jede `IClass` haben die Möglichkeit Z-Definitionen in einem `IZTypeDefinition`-Objekt zu definieren.

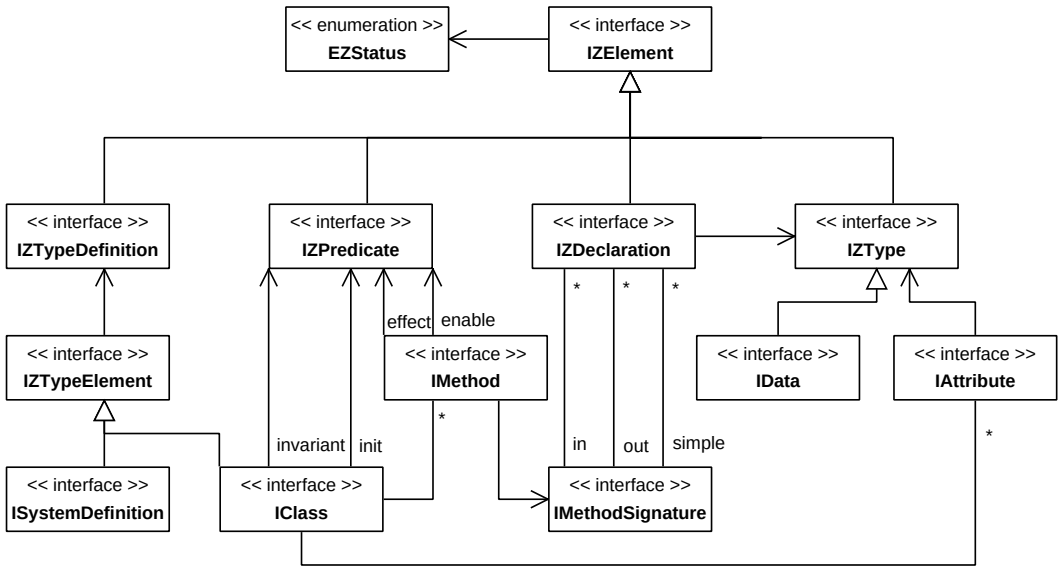


Abbildung 15.15: Z-Elemente

Kontext: Verbindungen

Das Superinterface aller Verbindungselemente ist `IConnection`. Folgende Interfaces sind ihr untergeordnet.

- Verbindungen der *Model-Ebene*:
  - `IClassConnection` als Ober-Interface für alle Verbindungen, die in Klassendiagrammen vorkommen können. Dies sind im Einzelnen: `IAggregation`, `IAssociation`, `IComposition`, `IDependency`, `IGeneralization` und `IRealization`.
  - `ITransition`, als einzige Verbindung auf Modelebene, die nicht ins Klassendiagramm gehört.
- Verbindungen der *Viewable-Ebene*:
  - `IViewableConnection` als Oberklasse für alle Verbindungen dieser Ebene. Dazu gehören zum einen die Viewable-Elemente der Verbindungen auf *Model*-Ebene und zum anderen die Verbindungen, die keine Entsprechung im *Model* haben. Letztere werden im Abschnitt *Kontext: Diagramme und Verbindungen auf Viewable-Ebene*, S. 177 näher erläutert.
- Die (einzige) Verbindung der *View-Ebene*:

– `IEdge`. Jede `IViewableConnection`-Instanz hat eine 1:1 Beziehung zu einer `IEdge`-Instanz. Näheres zur `IEdge` im Abschnitt *Kontext: Die View-Ebene*, S. 179.

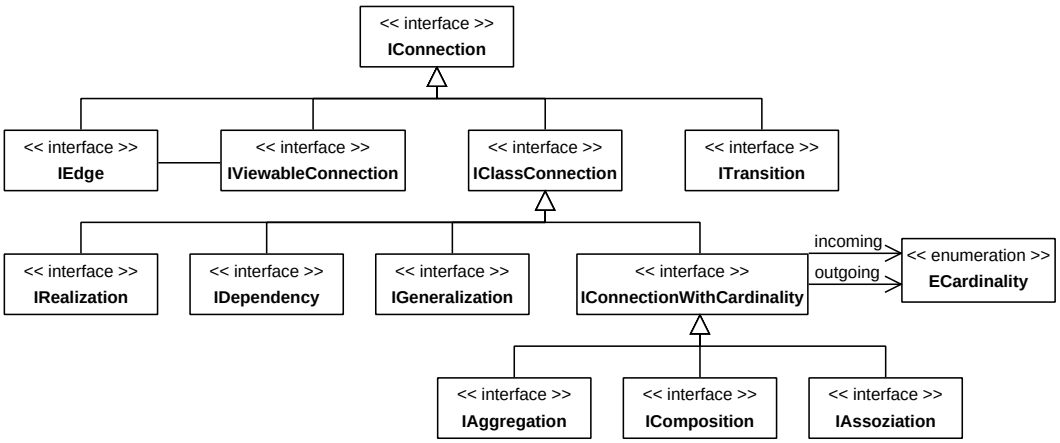


Abbildung 15.16: Verbindungen

Manche Verbindungen haben Kardinalitäten, die durch das Enum `ECardinality` repräsentiert werden. Dies sind im Einzelnen `IAggregation`, `IComposition` und `IAssociation`.

Kontext: Zustandsautomaten

Die Zustandsdiagramme von *Syspect* weisen im Vergleich zum Klassendiagramm zwei Besonderheiten auf, die sich auch in der Datenhaltung wiederspiegeln:

- Da es möglich ist, eine *Region* eines Zustands in einem eigenen Diagramm zu betrachten, gibt es zwei Arten von Zustandsdiagrammen: Solche, die einen vollständigen Zustandsautomaten enthalten und solche, die nur eine Region eines bestimmten Zustands zeigen.

- Während ein Klassendiagramm eine reine *View*-Angelegenheit ist (Klassen können in keinem, einem einzigen oder mehreren Diagrammen angeordnet werden und jeweils in verschiedene Kontexte gestellt werden), ist ein Zustandsautomat immer genau einer Klasse zugehörig und eine Klasse kann nicht mehrere davon haben.

`IStateMachine` ist das Modelelement für die Diagrammart *Zustandsdiagramm*. Hier ist zu beachten, dass dies die einzige Diagrammart ist, bei der auch das Diagramm selbst ein Modelelement hat. `IStateMachine` und `IRegion` erben beide von `IStateContainer`, welche `IAbstractStates` enthalten.
Es gibt drei Arten von Zuständen, die sich alle von `IAbstractState` ableiten:

- `IState` ist der „normale“ Zustand. Ein Zustand kann `IRegions` enthalten.

- `IInitState` ist der Startzustand.
- `IFinalState` ist der Endzustand.

`IAbstractState`-Objekte können `ITransitions` als ankommende oder abgehende Verbindungen haben. Eine `ITransition` kann eine `MethodSignature` als `trigger` besitzen.

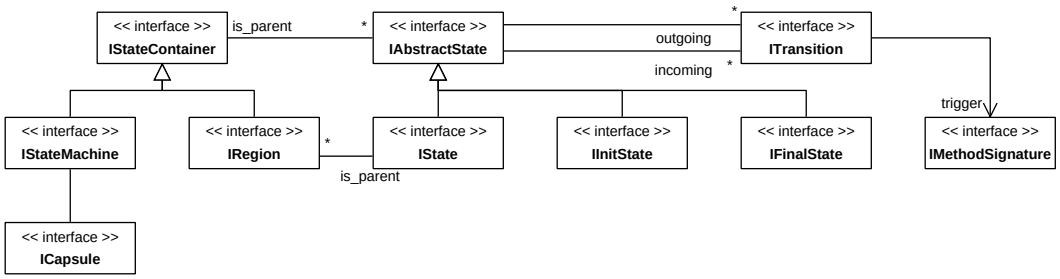


Abbildung 15.17: Zustandsautomaten

Kontext: Komponentendiagramme

In diesem Abschnitt werden alle Elemente beschrieben, die für Komponentendiagramme wichtig sind. Generell zu beachten ist, dass hier alle Klassen auf der *Viewable-Ebene* angesiedelt sind und es für sie keine eigenen Klassen auf der reinen *Model*-Ebene gibt.

- `IComponent` repräsentiert eine Komponente. Eine Komponente ist gewissermaßen eine `Capsule`-Instanz, bzw. jede Komponente bseitzt eine *Capsule* als *Model*.
- Eine Komponente kann mehrere `IPorts` haben.
- `IPort` und `IComponent` erben beide von `IConnectorContainer`. Dies bedeutet, sie können beide über `IConnectorConnection` mit `IConnector`-Instanzen verbunden sein. Sie verwalten aber auch direkt die Instanzen der Konnektoren, mit denen sie verbunden sind. Diese redundante Datenhaltung ist dem Umstand geschuldet, dass die `IConnectorConnection` für die grafische Darstellung relevant ist, für den Export jedoch nicht – deswegen ist für letzteren sozusagen die Möglichkeit eingebaut, direkt die Konnektoren abfragen zu können.
- `IConnector` ist das Superinterface für `IProvidedInterface` und `IRequiredInterface`. Ein `IConnector` kann über ein `IDelegate` mit einem `IPort` verbunden sein.

  Ein `IConnector` kennt immer sein `IInterface`, dass durch ihn bereitgestellt (*provided*) oder von ihm benötigt (*required*) wird.
- `IAbstractConnector` ist das Superinterface von `IConnector` und `IAssemblyConnector`.
- `IComponentContent` ist das Superinterface von `IComponent` und `IConnector`. Es ist die Schnittstelle für Elemente, die Kindelemente von Komponenten seien können und damit eine entsprechende Beziehung wiederum zu `IComponent`.

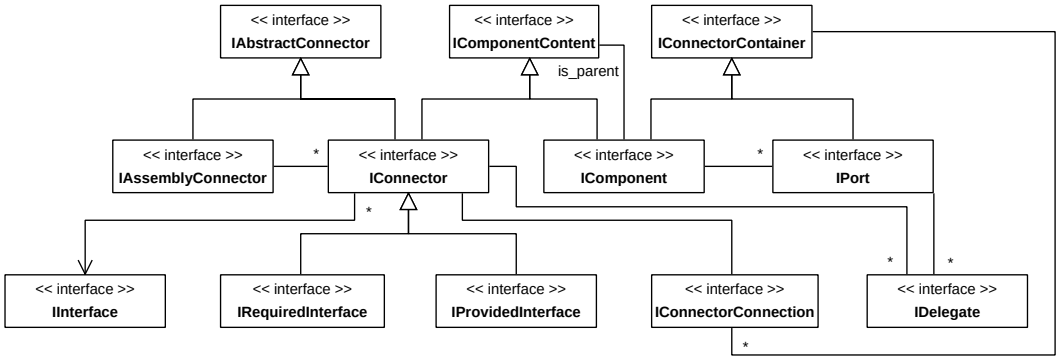


Abbildung 15.18: Komponentendiagramme

Kontext: Abstrakte Oberstruktur der *Viewable*-Ebene

Die *Viewable-Ebene* hat folgende Struktur: Alle Klassen dieser Ebene erben von der Wurzelklasse `IViewable`. Auf der darunter liegenden Ebene gibt es zunächst für jede Basis-Objektart (Diagramm, Verbindung, Knoten) eine eigene Oberklasse: `IViewableDiagram`, `IViewableConnection`, `IViewableNode`. Zusätzlich wurde ein Interface `IViewableWithModel` eingeführt, da alle Elemente dieser Ebene aufteilbar in zwei Gruppen sind: In der einen Gruppe haben alle Elemente ein dazugehöriges Element auf der reinen *Model-Ebene* (wie `IVCapsule` und `ICapsule`) und in der anderen Gruppe haben alle Elemente so ein Element auf der reinen *Model-Ebene* eben nicht (wie `IPort`). Alle Elemente, die zur ersten Gruppe gehören, implementieren besagtes Interface.

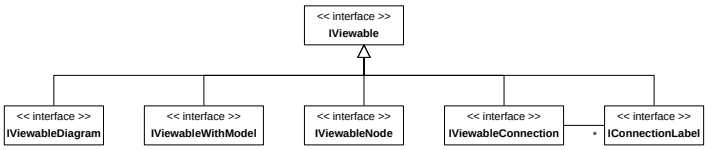


Abbildung 15.19: Abstrakte Oberstruktur der *Viewable*-Ebene

Kontext: Diagramme und Diagramminhalt

Damit ein Diagramm, unabhängig von seinem Typ, mit `getChildren` nach seinem Inhalt gefragt werden kann, erben alle Diagrammtypen von `IViewableDiagram` und die jeweiligen Hauptinhalte (von ihnen aus sind über weitere `get`-Methoden alle weiteren Inhalte zugreifbar) von `IDiagramContent`.

Kontext: Diagramme und Verbindungen auf *Viewable*-Ebene

Passend zu den von *Syspect* unterstützten Diagrammarten sind die drei erbenden Interfaces von `IViewableDiagram` `IComponentdiagram`, `IClassdiagram` und

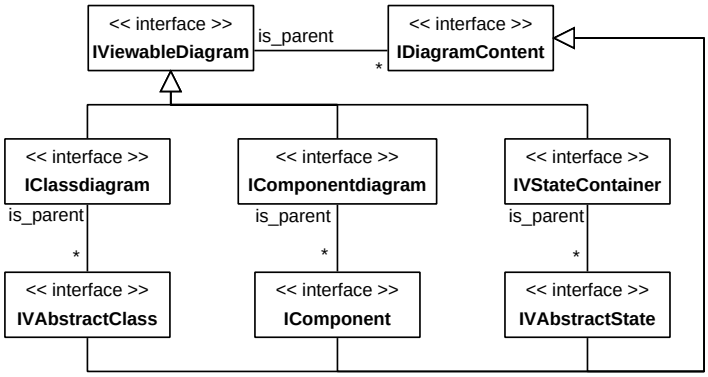


Abbildung 15.20: Kontext: Diagramme und Diagramminhalt

`IVStateContainer`. Letzteres Interface erbt ebenfalls von `IViewableWithModel` und hat als erbende Kinder `IVRegion` und `IVStateMachine.`
Die erbenden Interfaces von `IViewableConnection` lassen sich in zwei Gruppen einteilen:

- `IVTransition` und `IVClassConnection` haben korrespondierende Interfaces auf der *Model-Ebene* und erben deswegen auch von `IViewableWithModel`.
- `IConnectorConnection` und `IDelegate` haben keine zugehörigen Modelelemente.

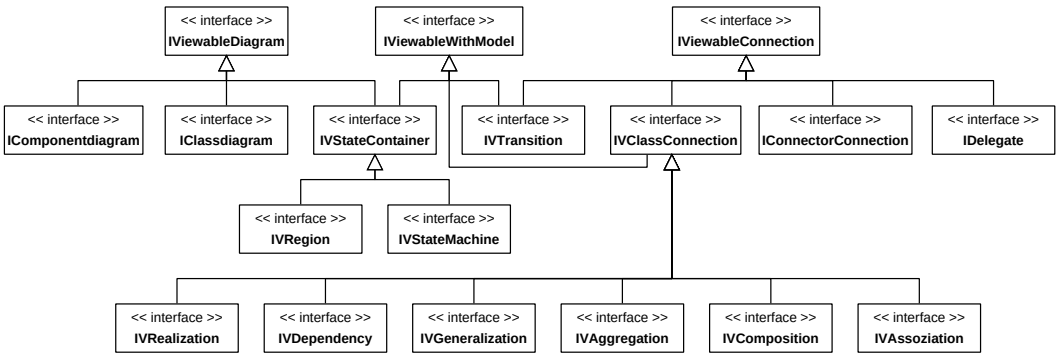


Abbildung 15.21: Diagramme und Verbindungen auf *Viewable*-Ebene

Kontext: Die Vererbungshierarchie von `ViewableNode`

Ebenso wie bei `IViewableConnection` lassen sich die erbenden Interfaces von `IViewableNode` in zwei Gruppen einteilen:

- `IVClassContent`, `IVAbstractClass` und `IVAbstractState` haben zugehörige Interfaces auf der *Model-Ebene*.
- `IConnectorContainer` und `IAbstractConnector` sind nur auf der *Viewable-Ebene* angesiedelt.

Dementsprechend leiteten sich die Elemente der ersten Gruppe zusätzlich von `IViewableWithModel` ab, die der zweiten Gruppe hingegen nicht.

Es sei darauf hingewiesen, dass alles ein `IViewableNode` ist, was in einem Diagramm gezeichnet werden kann und keine Verbindung ist. Die Unterscheidung zwischen Elementen, die eine eigene Position und Größe haben und solchen, deren Position und Größe direkt von anderen abhängig ist, wird erst auf *View*-Ebene getroffen und entsprechend im folgenden Abschnitt behandelt.

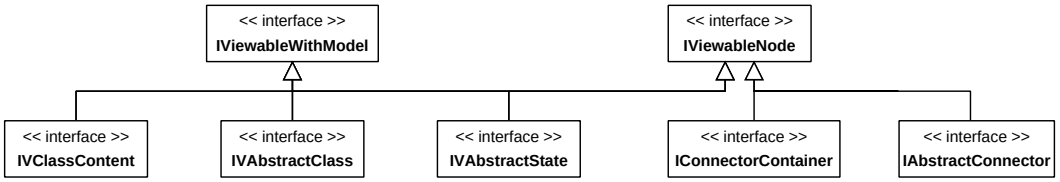


Abbildung 15.22: Die Vererbungshierarchie von `ViewableNode`

Kontext: Die *View*-Ebene

Die *View-Ebene* ist für die rein grafische Notation verantwortlich. Alle Interfaces dieser Ebene erben von `IView`. Mit `IDiagram`, `INode` und `IEdge` ergibt sich zunächt eine korrespondierende Struktur zu den abstrakten Viewable-Elementen (siehe Abschnitt 15.3.6, Seite 177). Mit `IConstrainedNode` liegt eine Spezialisierung von `INode` vor die eine Position und Größe in einem Diagramm beinhaltet.

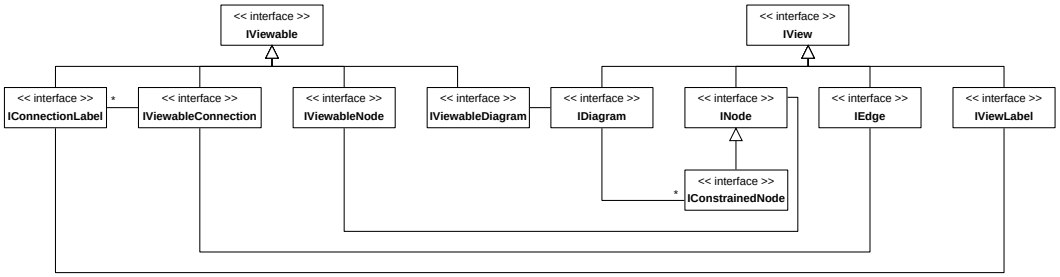


Abbildung 15.23: Die *View*-Ebene

15.3.7 SyspectModelConsistencyChecker Plug-in

Das ModelConsistencyChecker Plug-in überprüft, angestoßen durch ein Event, das aktuelle Projekt auf Inkonsistenzen. Diese werden in der `ProblemView` angezeigt und sind in die zwei Problemarten *Error* und *Warning* unterteilt. Als zusätzliche Besonderheit werden die Elemente, welche ein Problem ausgelöst haben, farbig markiert.

Schnittstellen

Ein Aufruf der Methode `notifyChanged` des `ProjectChangedNotifiers` im Core Plug-in wirft das Event `PROJECT_CHANGED`. Dieses führt neben dem zu überprüfenden Projekt auch ein Set von Elementen mit sich, welche sich seit dem letzten Aufruf verändert haben.
Der ModelConsistencyChecker reagiert mit dem Start der Konsistenzüberprüfung.
Der SyspectModelConsistencyChecker erweitert die *Eclipse* Extension Points
`org.eclipse.ui.perspectiveExtensions`,
`org.eclipse.ui.startup` und
`org.eclipse.ui.views`, welche im Abschnitt 15.2.2 (S. 154) beschrieben werden.

Weitere Abhängigkeiten existieren zu den Plug-ins
`de.syspect.componenttable`,
`de.syspect.core`,
`org.eclipse.core.runtime` und
`org.eclipse.ui.views`.

Ablauf der Konsistenzüberprüfung

Der gesamte Vorgang gliedert sich in drei Teile. Als erstes werden Referenzen für das aktuelle Projekt gesetzt und Container für den schnellen Zugriff auf Components und Klassen erstellt. Dies dient der Vorbereitung für den zweiten Schritt, in welchem die eigentlichen Konsistenzbedingungen überprüft und betroffene Elemente zwei internen Listen hinzugefügt werden. Neben dem Element werden den Listen noch eine genaue Fehlerbeschreibung und der Typ des Problems, *Error* oder *Warning*, mitgeliefert. Eine der Listen dient dazu die ProblemView zu aktualisieren. Sämtliche Elemente in der anderen Liste werden entsprechend des Problemtyps mit Hilfe der Methode `addProblem` als problematisch markiert und in vorkommenden Diagrammen eingefärbt. Diese Synchronisation mit der Anwendungsoberfläche ist als dritter Teil zu verstehen.

15.3.8 SyspectDiagramEditorBundle Plug-in

Dieses Plug-in ist dafür zuständig, gemeinsam genutzte Funktionalität der Diagrammeditoren zu verwalten. Abbildung 15.24 macht an einem Beispiel deutlich, wie das *SyspectDiagramEditorBundle* im Kontext von *Syspect* eingebettet ist: Es werden drei Commands betrachtet, die für die Erstellung jeweils eines Elements (`Method`, `Association`, `Interface`) im Klassendiagrammeditor zuständig sind.
Die Klasse `Command` und das dazugehörige Prinzip wird von GEF bereitgestellt. Das im core-Paket definierte `SyspectCommand` leitet sich davon ab und erweitert das GEF-Konzept um ein *Syspect*-spezifisches Konzept.
Das *SyspectDiagramEditorBundle* stellt nun die Klassen `CreateConnectionCommand` und `CreateCommandWithConstraints` bereit, die von allen Diagrammeditoren benutzt werden können. Das Plug-in, das für den Klassendiagrameditor zuständig ist, kann nun auf dieser Hierarchie aufbauen – hier muss nur der Code implementiert werden,

der „Klassendiagramm-spezifisch“ ist.

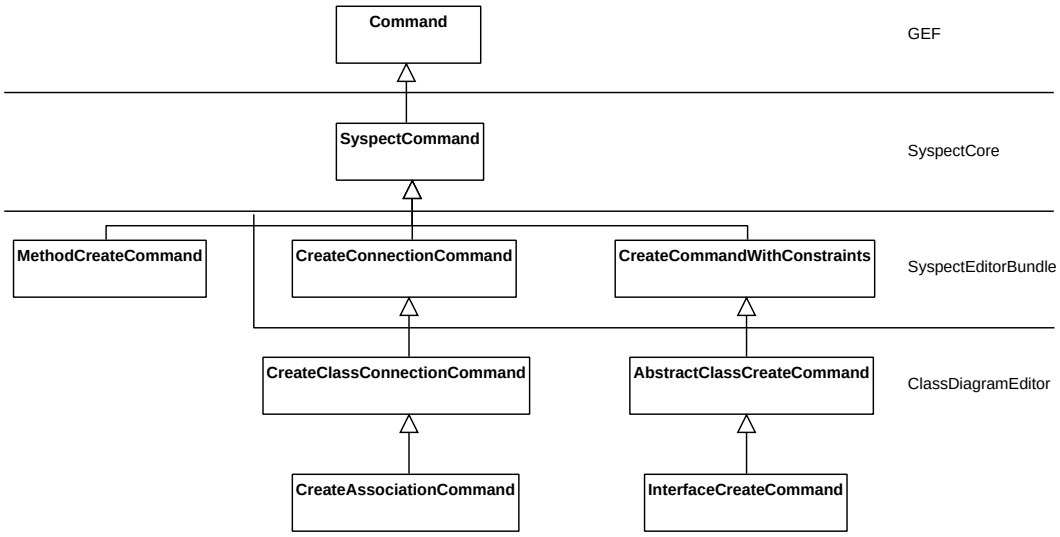


Abbildung 15.24: Vererbungshierarchie bei den CreateCommands

Nach demselben Prinzip sind in diesem Plug-in auch Klassenhierarchien anderer GEF-Grundtypen wie `EditPart`, `EditPolicy` und `Figure` (siehe Abschnitt 15.2.1, S. 149) bereitgestellt, um das Implementieren von Diagrameditoren zu erleichtern.
Da es – wie oben schon erwähnt – ein *Syspect*-eigenes Command-Konzept gibt, soll im folgenden Abschnitt näher darauf eingegangen werden.

Commands

Eine elementare Rolle spielt in *Syspect* das *Command Pattern*, denn jede Änderung am Model wird durch Commands ausgeführt. Die von uns verwendeten Commands bestehen aus folgenden Methoden, deren Inhalte im Folgenden beschrieben werden:

- In den Konstruktoren sollen nur die übergebenen Felder gespeichert werden, da der Konstruktor potentiell sehr oft aufgerufen wird.

- `init`: Hier soll das Erstellen von Objekten, z.B. eines Teilbaums mit einem *Viewable*- und einem *View*-Modelelement, geschehen. Wichtig ist hierbei, dass keine Referenzen der erstellten Objekte an das Modell übergeben werden. Es darf also auf einem erstellten Objekt beispielsweise `setParent` mit einem bestehenden Modell-Objekt als Parameter aufgerufen werden, jedoch ist ein `addChild` auf dem bestehenden Objekt verboten.

- `canExecute`: Die `canExecute`-Methode dient für die Umgebung dazu, herauszufinden ob das Command ausgeführt werden darf oder nicht. Sie greift in der Regel auf Consistency-Checker-Methoden zurück um dies herauszufinden.

- `execute`: Die `execute`-Methode sollte nicht verwendet werden, sie wird benutzt um das Command in den Undo-Manager einzufügen, darüberhinaus ruft sie `reproduce` auf.

- `reproduce`: Die `reproduce`-Methode nimmt die eigentlichen Änderungen am Modell vor. Hier werden Namen, mithilfe des `NameAdministrators` erstellt, sowie Änderungen an bestehenden Modell-Elementen vorgenommen. Ein Beispiel für eine typische Anweisung, die in `reproduce` geschehen sollte ist `addChild` um einen neuerstellten Teilbaum in das Modell einzufügen. Wichtig ist, dass die `reproduce`-Methode auf jeden Fall `super.reproduce` aufruft, denn sonst wird `init` nicht ausgeführt.

- `undo`: Die `undo` Methode macht nun genau das rückgängig, was die `reproduce` Methode gemacht hat. An dieser Definition ist auch zu sehen, dass die `reproduce`-Methode keine Objekte erstellen sollte, denn dies ist in Java nicht explizit rückgängig zu machen.

- `toString`: Als letztes gibt es noch die `toString`-Methode, die einen Text liefern soll, der bei den Undo/Redo-Buttons und Menüeinträgen angezeigt wird.

Zum *Command Stacking*, also dem Benutzen von Commands innerhalb von Commands, ist zu bemerken, dass die Commands dann nur erzeugt und `redo` beziehungsweise `undo` delegiert werden muss. Die `execute`-Methode darf nicht aufgerufen werden, weil andernfalls die Commands alle in den Undo-Manager eingetragen würden, was ein atomares Undo verhindern würde.

Schnittstellen

Extension Points:

- `componentdiagramconsistency` Das Plug-in, welches die Benutzereingaben in Komponentendiagrammen prüfen soll, muss das Interface `IComponentDiagramConsistencyChecker` implementieren.

- `statemachineconsistency` Das Plug-in, welches die Benutzereingaben in Zustandsautomaten prüfen soll, muss das Interface `ICStatemachineConsistencyChecker` implementieren.

- `classdiagramconsistency` Das Plug-in, welches die Benutzereingaben in Klassendiagrammen prüfen soll, muss das Interface `IClassDiagramConsistencyChecker` implementieren.

Extensions:

- `de.syspect.preferenceinitializer` (siehe Abschnitt 15.3.2, S. 158)

- `org.eclipse.help.toc` (siehe Abschnitt 15.2.2, S. 154)

- `org.eclipse.ui.preferencePages` (siehe Abschnitt 15.2.2, S. 154)

Dieses Plug-in hat folgende weitere Abhängigkeiten:

- `org.eclipse.core.runtime`
- `org.eclipse.equinox.common`
- `org.eclipse.gef`
- `org.eclipse.ui.views`

15.3.9 Die drei Diagrammeditoren

Die Beschreibung der drei Diagrammeditor-Plug-ins ist einem Kapitel zusammengefasst, da sie viele Gemeinsamkeiten aufweisen – zum Beispiel stimmen die meisten Schnittstellen überein.
Jedes dieser Plug-ins erweitert *Syspect* um einen grafischen Editor für jeweils einen der drei Diagrammtypen, die erforderlich sind, um die Anforderungen des UML-Profils zu erfüllen (siehe Kapitel 9, S. 77 und Kapitel **??**, S. **??**). Sie verwenden hierfür sowohl das GEF-Plug-in, als auch das *Syspect*-eigene `de.syspect.editorbundle`. Diese Architektur soll es erleichtern, weitere Diagrammtypen zu *Syspect* hinzufügen zu können.

Die Paketstruktur der Diagrammeditor-Plug-ins ist nach dem in Abschnitt 15.3.1 (S. 157) vorgestellten Prinzip in *Controller*- und *View*-Teil untergliedert. Während im *Controller*-Bereich sowohl die benötigten GEF-Klassen wie `EditParts`, `EditPolicies` und `Commands` als auch `Actions` und jeweils ein Sprachpaket untergebracht sind, werden im *View*-Bereich neben den GEF-`Figures` auch die *View*-Anteile der *Wizards* und *Preferences* bereitgehalten.

In den folgenden Abschnitten werden pro Plug-in einige Besonderheiten und Techniken vorgestellt, die für den Entwurf des Editors für die jeweilige Diagrammart zu beachten wichtig sind.

SyspectClassdiagramEditor Plug-in

Für den Klassendiagrammeditor sind zwei verschiedene GEF-Layout-Konzepte notwendig, um die grafischen Elemente richtig anordnen zu können:

- Mithilfe einer `XYLayoutEditPolicy` werden die Klassen und Interfaces innerhalb des Diagramms angeordnet. Dieses Konzept wird immer dann angewendet, wenn ein grafisches Element innerhalb eines anderen größeren in X- und Y-Richtung frei verschiebbar sein soll.
- Den Effekt, dass Attribute und Methoden als eine Art Liste innerhalb der Klasse erscheinen, wird hingegen mit der `FlowLayoutEditPolicy` realisiert.

Um Klassen und Interfaces das UML-typische Aussehen zu verleihen, sind neben dem angezeigten Rechteck und der Attribut/Methoden-Liste der Name und separierende

Linien notwendig. Da hierfür einerseits keine eigene Datenhaltung notwendig ist (die Klasse speichert ihren Namen selbst, die Separatorpositionen werden vom *Controller* bei jedem Zeichnen neu berechnet), aber andererseits das GEF-Framework für jedes zu zeichnende Element ein dahinterstehendes *Model* benötigt, haben wir uns für einen Kompromiss entschieden:
In der Klasse `SyspectNodeEditPart` werden entsprechende *Model*-Elemente als innere Klassen bereitgehalten. Dies ist für das GEF-Framework ausreichend, hat aber den Vorteil, dass diese Klassen nicht in unserer Datenhaltung vorkommen.
Durch Vererbung stehen diese inneren Klassen allen betreffenden EditParts zur Verfügung (siehe Abbildung 15.25).

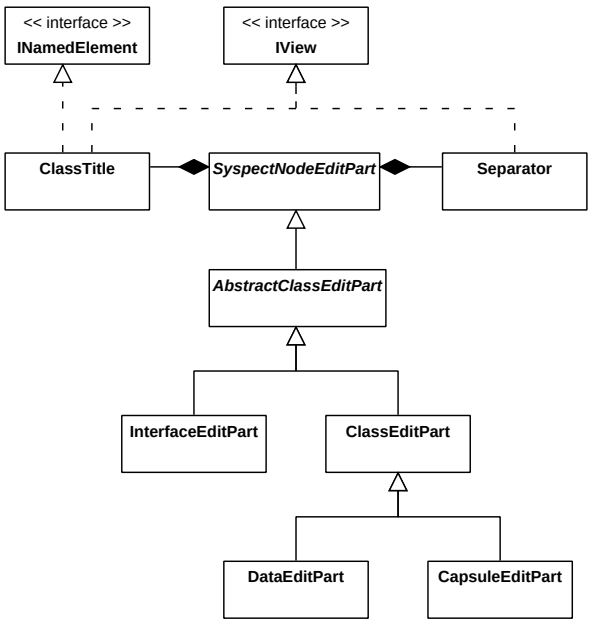


Abbildung 15.25: Bereitstellung von `ClassTitle` und `Separator`

SyspectComponentdiagramEditor Plug-in

Herausforderungen bei der Entwicklung des Komponentendiagrammeditors waren vor allem Eigenschaften des Komponentendiagramms, welche nicht typisch für das GEF-Konzept sind.
Hierunter fallen unter anderen die Hauptzeichenelemente *Provided/Required Interfaces*, der *AssemblyConnector* und der *Port*.

- Ein Provided/Required Interface wird im Editor durch eine Figure und eine Verbindungslinie zu einer Komponente dargestellt. Demnach müsste erst eine Provided/Required Figure im Diagramm gezeichnet werden, die anschließend mit einer Komponente durch eine Verbindungslinie verbunden wird. Da allerdings ein Provided/Required Interface immer zu einer Komponente gehört, sollte die Figure und die Verbindung in einem gezeichnet werden. D.h. ein

Provided/Required Interfaces, und nicht nur die Verbindung, muss einer Komponente zugewiesen werden. Dieses wurde so realisiert, dass erst eine Verbindung durch den Command `CreateProvide/RequireWithWizardCommand` gezeichnet wird und dann automatisch die Figure des jeweiligen Interfaces als Target der Verbindung durch den Command `ProvidedCreateCommand` gesetzt wird. Hierzu ruft der `CreateProvide/RequireWithWizardCommand` den Command `ProvidedCreateCommand` auf. Die Abbildung 15.26 zeigt diesen Vorgang anhand von Screenshots des Editors.

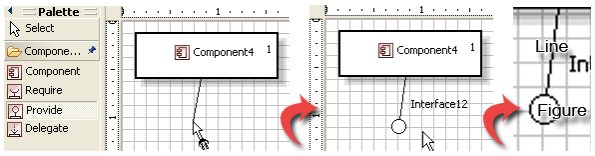


Abbildung 15.26: Die Erstellung eines Provided Interfaces bestehend aus einer Figure und einer Verbindungslinie

Eine weitere Schwierigkeit bestand darin, einem Provided/Required Interface ein Modelelement zuzuweisen, da hier auch schon bestehende Modelelemente verwendet werden können und nicht immer ein neues erzeugt werden soll. Zu diesem Zweck wurde in den `CreateConnectorWithWizardCommand` ein kompletter Wizard eingebaut, der bei der Ausführung des Commands aufgerufen wird und eine Auswahl an möglichen Modelelementen zur Verfügung stellt.

Des Weiteren gehört zwar ein Provided/Required Interface zu einer Komponente, allerdings wird die Figure außerhalb der Komponente gezeichnet. Dadurch ergibt sich der Konflikt, dass im Model ein Interface als Child einer Komponente gespeichert werden muss, im Editor aber nicht als Child gezeichnet werden kann, da die Figure außerhalb des Zeichenbereichs der Komponente liegt. In Abbildung 15.27 ist der Zeichenbereich eine Komponente durch ein rotes Gitter gekennzeichnet. Die innere Komponente kann problemlos als Kind der äußeren Komponente gespeichert und gezeichnet werden. Die Figure des Provided Interfaces liegt jedoch außerhalb, wodurch sie nicht als Kind von ihrer zugehörigen Komponente gezeichnet werden kann. Dieses musste im Editor so umgangen werden, dass ein Parent von einer Komponente dessen Interfaces zeichnet, d.h. im Detail der EditPart des Komponentendiagramms und der einer Komponente zeichnen ihre Komponenten-Kinder und auch deren Interfaces.

- Ein Provided Interface und ein Required Interface können miteinander verbunden werden, wodurch sich ein *AssemblyConnector* ergibt. Hierbei tritt die Schwierigkeit auf, dass zwei Figuren miteinander verbunden werden müssen, die keine Verbindungslinie haben und dass beide Figuren durch einen AssemblyConnector ersetzt werden müssen. Hierbei wurde eine Policy verwendet, mit der man Figuren zu Figuren hinzufügen kann - also ein Provided zu einem Required Interface

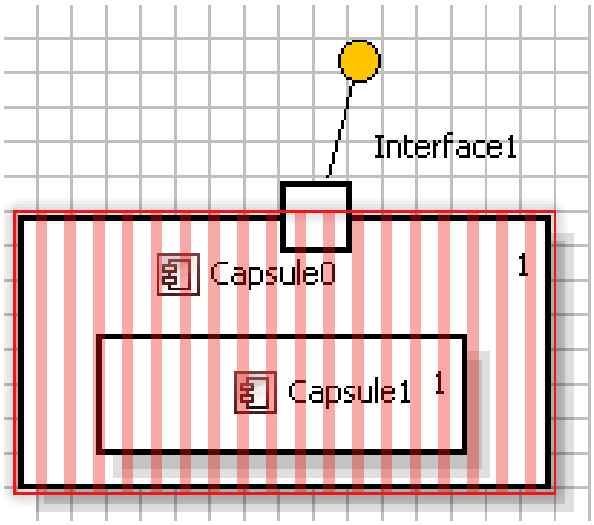


Abbildung 15.27: Der Zeichenbereich einer Komponente rot hinterlegt.

und umgekehrt (ConnectorXYLayoutPolicy). Der dadurch aufgerufene Command `AssemblyConnectorCreateCommand` fügt beiden Interfaces ein AssemblyConnector Modelelement hinzu. Bei dem Zeichnen der Interfaces wird im entsprechenden EditPart darauf getestet, ob ein Interface einen AssemblyConnector hat und dieser dann anstelle des Interfaces `Figure` gezeichnet. D.h. für die Verbindungslinie ist der neue Target der AssemblyConnector, wobei hier die Ausrichtung der Verbindungslinie ebenfalls eine Schwierigkeit darstellt. Die Verbindungslinie sollte an dem Teilbereich der AssemblyConnector Figure enden, die das damit voher verbundene Interface darstellt. Also sollte die Verbindungslinie des Provided Interfaces auch an der Stelle des AssemblyConnectors enden, die das Provided Interface darstellt. Sonst wäre eine Unterscheidung, ob eine Komponente ein Required oder Provided Interface hat nicht möglich, da der AssemblyConnector keine Unterscheidung macht. Dieses wurde versucht mit dem von GEF zur Verfügung gestellten Konzept der RotatableDecoration zu lösen. Die RotatableDecoration dreht die Target Figure einer Linie mit, wenn sich die Position der Verbindung ändert. Dadurch kann die Verbindung einen festen Ankerpunkt am Target beibehalten. Allerdings wurde das Problem nicht vollständig gelöst. Es lassen sich die Figuren im Editor so anordnen, dass zum Beispiel eine Verbindung eines Provided Interfaces an dem Bereich der Figure des AssemblyConnetor endet, der das Required Interfaces darstellt (siehe Abbildung 15.28). Hier bedarf es wahrscheinlich der Erstellung eines eigenen ConnectionAnchors für den AssemblyConnector der diese Eigenschaft erfüllt.

- Eine weitere Figure des Komponentendiagramms, welche ebenfalls nicht trivial zu implementieren war, ist der *Port*. Ein Port hat die Eigenschaft, dass dieser sich nur auf der Kante einer Komponente befindet und somit innerhalb und außerhalb der Komponente gezeichnet werden muss. Hierbei tritt, wie oben schon bei dem Required/Provided Interfaces beschrieben, wieder die Schwierigkeit der

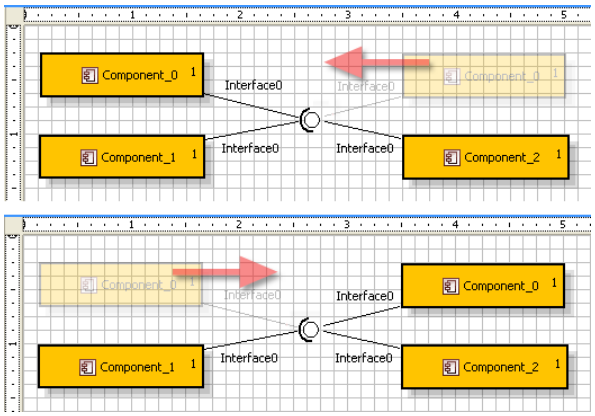


Abbildung 15.28: Der AssemblyConnector macht keinen Unterschied zwischen Provided oder Required Interfaces

Parent Child Beziehung zwischen Model und Diagramm auf. Dieses wurde nach dem gleichen Prinzip gelöst, welches schon bei den Provided/Required Interfaces beschrieben worden ist und wird deshalb nicht noch einmal erwähnt.
Allerdings kommt bei dem Port noch die Ausrichtung auf der Kante einer Komponente hinzu. Wenn ein Port verschoben wird, dann soll dieser sich automatisch wieder auf der ihm am nächsten liegenden Kante seiner Komponente ausrichten. Das wurde dadurch realisiert, dass in dem Command, der den Port im Diagramm verschiebt (`PortChangeConstraintCommand`), die neue Position des Port durch Berechnungen so angepasst wird, dass dieser auf einer Kante der Komponente liegt.
Da ein Port nur erzeugt werden soll, wenn eine Komponente eine innere Komponenten hat, wurde der Port als Zeichenelement aus der Palette des Editors entfernt. Wenn ein Port im Diagramm benötigt wird, wird dieser automatisch gesetzt. Die Implementierung für das automatische Setzen des Ports wurde in den `ConnectorCreateCommand` eingefügt, da ein Port den Source des Connector bildet. Aus diesem Command wird im Falle von inneren Komponenten der Command `PortsCreateCommand` aufgerufen, der einen neuen Port erstellt.

SyspectStatemachineEditor Plug-in

Besondere Anforderungen an den Editor für Zustandsautomaten waren:

- Die durch das UML-Profil vorgesehene Möglichkeit der Schachtelung von Zustandsautomaten mithilfe von Regionen in Zuständen.

- Die Möglichkeit, dass eine Transition denselben Zustand als Quelle und Ziel hat.

  Dies wurde durch das Erstellen zweier zusätzlicher *Bendpoints* im `TransitionCreateCommand` gelöst, die in die View-Repräsentation – also in

ihre `IEdge`-Instanz – eingeflochten werden, so dass die Transition entsprechend umgebogen wird und wieder auf ihren Startzustand zeigt.

Schnittstellen

Alle drei Diagrammeditor-Plug-ins sprechen dieselben Extension Points an:
Es werden vier Core Extension Points (`de.syspect.core.preferenceinitializer`, `de.syspect.core.syspecteditpartfactory`, `de.syspect.core.syspectdeleteactionhook`, `de.syspect.core.syspectdiagram`) und zwei Navigator Extension Points (`de.syspect.navigator.actioncontributor`, `de.syspect.navigator.toolbarcontributor`) erweitert. Die Beschreibung dieser Extension Points können im Abschnitt 15.3.2 (S. 158) und im Abschnitt 15.3.3 (S. 163) nachgelesen werden.
Auf die *Eclipse* Extension Points die diese Plug-ins erweitert wurde bereits eingegangen (siehe Abschnitt 15.2.2, S. 154):
`org.eclipse.help.toc`, `org.eclipse.ui.actionSets`, `org.eclipse.ui.bindings`, `org.eclipse.ui.commands`, `org.eclipse.ui.editors`, `org.eclipse.ui.popupMenus` und `org.eclipse.ui.preferencePages`.

Die gemeinsamen Abhängigkeiten der drei Editor-Plug-ins sind:
`de.syspect.navigator`, `de.syspect.core`, `de.syspect.editorbundle`, `de.syspect.utils`, `org.eclipse.core.runtime`, `org.eclipse.gef`, `org.eclipse.ui` `org.eclipse.ui.views` und `org.eclipse.ui.views.porperties.tabbed`.
Zusätzlich ist das ClassdiagramEditor Plug-in von `org.eclipse.draw2d` und `de.syspect.ozdcutil` abhängig und das ComponentdiagramEditor Plug-in von `de.syspect.classdiagram`.

15.3.10 SyspectDiagramConsistencyChecker Plug-in

Das DiagramConsistencyChecker Plug-in stellt Methoden zur Validierung von Benutzereingaben bereit. Eine Rückmeldung bei falscher Eingabe bekommt der Benutzer umgehend durch die integrierte `ErrorView` mitgeteilt.
Der DiagramConsistencyChecker selbst ist in die drei Klassen `ClassDiagramConsistencyChecker`, `ComponentDiagramConsistencyChecker` und `StatemachineConsistencyChecker` aufgeteilt, welche die gemeinsame Oberklasse `ConsistencyChecker` besitzen und Design-Pattern *Singleton* implementieren.

Schnittstellen

Aufgerufen werden die DiagramConsistencyChecker üblicherweise in den `canExecute`-Methoden der das Model ändernden Commands. Diese Vorgehensweise liegt insofern auf der Hand, als dass so die DiagramConsistencyChecker die Kontrolle über die

Ausführung der Commands haben, und Modeländerungen verhindern können, die zu Inkonsistenzen führen würden. Der Zugriff erfolgt über die im EditorBundle bereitgestellten Extension Points:
`de.syspect.editorbundle.classdiagramconsistency`,
`de.syspect.editorbundle.componentdiagramconsistency` und
`de.syspect.editorbundle.statemachineconsistency` (siehe Abschnitt 15.3.8, S.182).

Eclipse-Extension Points werden ebenfalls erweitert:
`org.eclipse.help.toc`, `org.eclipse.ui.perspectiveExtensions` und
`org.eclipse.ui.views` (siehe Abschnitt 15.2.2, S. 154)

Das Plug-in hat weitere Abhängigkeiten zu folgenden Plug-ins:
`de.syspect.core`,
`org.eclipse.core.runtime`,
`org.eclipse.draw2d` und
`org.eclipse.ui.views`.

Prinzip anhand eines Beispiels

Vor dem Hinzufügen einer *Dependency* zwischen zwei Klassen soll geprüft werden, ob dies überhaupt möglich ist:

1. In der `canExecute`-Methode des Commands `DependencyCreateCommand` wird mit Hilfe der Klasse `Activator` die Referenz des aktuellen ClassDiagramConsistencyCheckers geholt und auf diesem die Methode `addDependency(IConnected vsource, IConnected vtarget)` aufgerufen.

2. Die Methode überprüft nun anhand der übergebenen Referenzen, ob eine *Dependency* zwischen den Klassen erlaubt ist. Konkret werden hier zunächst die Referenzen auf Gleichheit mit `null` geprüft. Sind die Referenzen vorhanden, wird validiert, dass eine Capsule als Quelle und ein Interface als Ziel vorliegt und noch keine *Dependency* zwischen beiden vorhanden ist. Weiterhin wird kontrolliert, ob Generalisierungsbeziehungen zu einer Inkonsistenz führen können.

3. Bei jedem der oben durchgeführten Tests wird sofort mit `return false` abgebrochen, sobald eine Konsistenzbedingung verletzt wurde. Außerdem wird der `ErrorView` mit `addErrorMessage` eine Beschreibung des Problems und eine Ursache hinzugefügt.

4. Wenn jedoch alle Tests bestanden werden, wird dem Command mit `return true` signalisiert, dass es ausgeführt werden darf.

15.3.11 Syspect OZ-DC-Util Plug-in

Das OZ-DC-Util stellt diverse Funktionen in Bezug auf Object-Z und den Duration Calculus zur Verfügung. Es kapselt die *Community Z Tools* mithilfe des `CZTWrappers`,

stellt einen Editor zum komfortablen Editieren von Formeln, diverse Listener, die zum Ersetzen von LaTeX-Eingaben des Benutzers in Unicodezeichen verwendet werden, sowie einen DC-Parser zum Umwandeln der eingegebenen DC Formeln in eine Java-Darstellung bereit. Letzter ist mithilfe von *javacc* generiert. Dieses Plugin ist aufgrund der vielfältigen Formel-Funktionalitäten ein zentraler Bestandteil unseres Tools und wird beispielsweise von dem Property-Plug-in, dem CSP-OZ-DC-, sowie dem PEA-Export benutzt.

Schnittstellen

Extensions:
`de.syspect.navigator.actioncontributor`

`org.eclipse.help.toc`,
`org.eclipse.core.runtime.preferences` und
`org.eclipse.ui.preferencePages`.

Weitere Abhängigkeiten:
`de.syspect.core`,
`de.syspect.utils`,
`org.eclipse.draw2d` und
`org.eclipse.ui.views`.

15.3.12 SyspectProperties Plug-in

Dieses Plug-in wird zur Erstellung des *Tabbed Property Views* benötigt, die für die Eigenschaftsseite des *Syspect*-Projektes verwendet wird. Die *Tabbed Property View* wurde von IBM für verschiedene Produkte, die auf *Eclipse* basieren, auf Open Source Basis entwickelt. Zuerst gehörte sie zur so genannten *Eclipse Web Tools Platform* und ab der *Eclipse*-Version 3.2 zur *Core Platform*. Implementiert ist sie im `org.eclipse.ui.views.properties.tabbed` Plug-In.

Schnittstellen

Extensions

Die *Tabbed Property View* wird über drei Extension Points konfiguriert (s. plugin.xml):

- *Property Contributor*
  (`org.eclipse.ui.views.properties.tabbed.PropertyConributor`)
- *Property Tabs*
  (`org.eclipse.ui.views.properties.tabbed.PropertyTabs`)
- *Property Sections*
  (`org.eclipse.ui.views.properties.tabbed.property.Sections`)

Abbildung 15.29: *Tabbed Property View*

Die einzelnen Extension Points, die von SyspectProperties implementiert werden, definieren eine Reihe von Attributen, auf die im Folgenden eingegangen wird.

Property Contributor

Der `Property Contributor` ist die Schnittstelle zwischen den einzelnen *WorkbenchParts* und einer *Tabbed Property View*, d.h. der `Contributor` wird von mehreren *WorkbenchParts* geteilt, die auf ein Model zugreifen wollen. Zu diesem Zweck müssen die entsprechenden *WorkbenchParts* das Interface `ITabbedPropertySheetPageContributor` implementieren. Dieses beinhaltet nur die Methode `getContributorId()`, über die die eindeutige ID des `Property Contributors` zurückgegeben wird. Des Weiteren muss den *WorkbenchParts* mitgeteilt werden, dass sie die *Tabbed Property View* benutzen sollen. Hierfür wird die Methode `getAdaptable(Class adapter)` überschrieben, die überprüft, ob der Adapter wirklich `IPropertySheetPage` ist und entsprechend die *Tabbed Property View* zurückgibt. Im *Syspect*-Projekt sind die relevanten *WorkbenchеParts* der Navigator, der ClassdiagramEditor, der ComponentdiagramEditor und der StatemachineEditor.

Optional können dem `Property Contributor` ein `TypeMapper` zum Abbilden der Typen von selektierten Objekten auf andere bzw. ein `LabelProvider` für die Darstellung hinzugefügt werden.

Property Tabs

Der `Property Tabs` Extension Point definiert die möglichen Reiter, die in der *Tabbed Property View* auftreten können. Über die `Contributor-ID` wird eine Referenz zum `Property Contributor` geschaffen. Jeder Reiter wird ebenfalls durch eine ID gekennzeichnet und erhält eine Beschriftung. Für die Anordnung der Reiter in der *View* gibt es das Attribut `afterTab`, das jedem Reiter mitteilt, nach welchem er folgt. Zusätzlich können die Reiter gruppiert werden.

```
<extension point="org.eclipse.ui.views.properties.tabbed.propertyTabs">

   <propertyTabs contributorId="syspectproperties.ID">
      <propertyTab
         id="syspectproperties.General"
         label="General"/>
   </propertyTabs>

</extension>
```

Abbildung 15.30: *Property Tabs* Extension Point

Abbildung 15.30 zeigt einen Ausschnitt aus der Konfigurationsdatei eines Plug-ins (plugin.xml), das den *Property Tabs* Extension Point definiert.

Property Sections

Über die `Property Sections` werden die `Widgets` für die *View* definiert. Diese enthalten eine ID und werden einem Reiter zugeordnet. Die Anordnung wird über das Attribut `afterSection` geregelt. Die `Widgets` werden in einer Klasse implementiert, die der `Section` zugeordnet wird. Diese Klasse wird von der Klasse `AbstractPropertySection` abgeleitet und implementiert die folgenden Methoden:

- `createControls()` erzeugt das eigentliche Widget
- Über `setInput()` wird der Workbench und das selektierte Modelelement übergeben
- `refresh()` aktualisiert die sichtbaren Bedienelemente

Die Aufrufreihenfolge durch das Framework entspricht der Reihenfolge der Auflistung. Die einzelnen `Sections` werden für bestimmte Eingabetypen des Models in die Reiter eingebunden. Hierfür werden die Typen über `input`-Attribute für die `Sections` definiert. Wenn für einen bestimmten Modeltyp keine `Section` in einen Reiter eingebunden wird, wird dieser ebenfalls nicht in die *View* eingebunden.
Abbildung 15.31 zeigt einen Ausschnitt der plugin.xml Datei, in der beispielhaft *Property Sections* Extension Point definiert wird.

Für das Syspect-Projekt werden diverse *Sections* bzw. *Section-Klassen* erzeugt. Das folgende Klassendiagramm (Abbildung 15.32) soll einen Überblick über die Struktur dieser Klassen geben.
Die einzelnen Elemente der Oberfläche sollen einem einheitlichen *Look and Feel* entsprechen. Zu diesem Zweck gibt es die `TabbedPropertySheetWidgetFactory`, die über die Methode `getWidgetFactory()` aufgerufen wird. Über `create`-Methoden können

```
<extension point="org.eclipse.ui.views.properties.tabbed
   .propertySections">

   <propertySections contributorId="syspectproperties.ID">
      <propertySection
         afterSection="syspectproperties.ElementNameSection"
         class="de.syspect.syspectproperties.view.MultiplicitySection"
         id ="syspectproperties.MultiplicitySection"
         tab="syspectproperties.General">
            <input type="de.syspect.modelinterface.viewable.IComponent"/>
            <input type="de.syspect.componentdiagrameditor.controller
               .editparts.ComponentEditPart"/>
      </propertySection>
   </propertySections>

</extension>
```

Abbildung 15.31: *Property Sections* Extension Point

dann die einzelnen Elemente erzeugt werden. Im *Syspect*-Projekt werden die Klassen `AbstractPropertySection` und `TabbedPropertySheetWidgetFactory` erweitert, um ein `StyledText`-Element erzeugen zu können. Dieses wurde für interne Zwecke bevorzugt.
Um auf Aktualisierungen des Models reagieren zu können, muss man die `Section` als `Listener` (PropertyChangeListener) beim *Model* registrieren (vgl. Abbildung 15.30). Dies ermöglicht immer die aktuelle Darstellung des Models in der *Tabbed Property View*.

Um andererseits Eingaben in der Tabbed Property View ins Model zu propagieren, wird ein `SimplePropertyChangeCommand` aufgerufen, dass mithilfe des Java-Beans-Property-Konzeptes und des Enums `EPropertyConstants` die Attribute des Models neu setzt.

Zum Einbinden von Hilfe-Seiten in *Syspect* erweitert dieses Plug-in ebenfalls `org.eclipse.help.toc`.

Dependencies

Das SyspectProperties Plug-in ist weiterhin von folgenden Plug-ins abhängig:

`de.syspect.core`,
`de.syspect.editorbundle`,
`de.syspect.navigator`,
`de.syspect.ozdcutil`,
`de.sypsect.utils`,

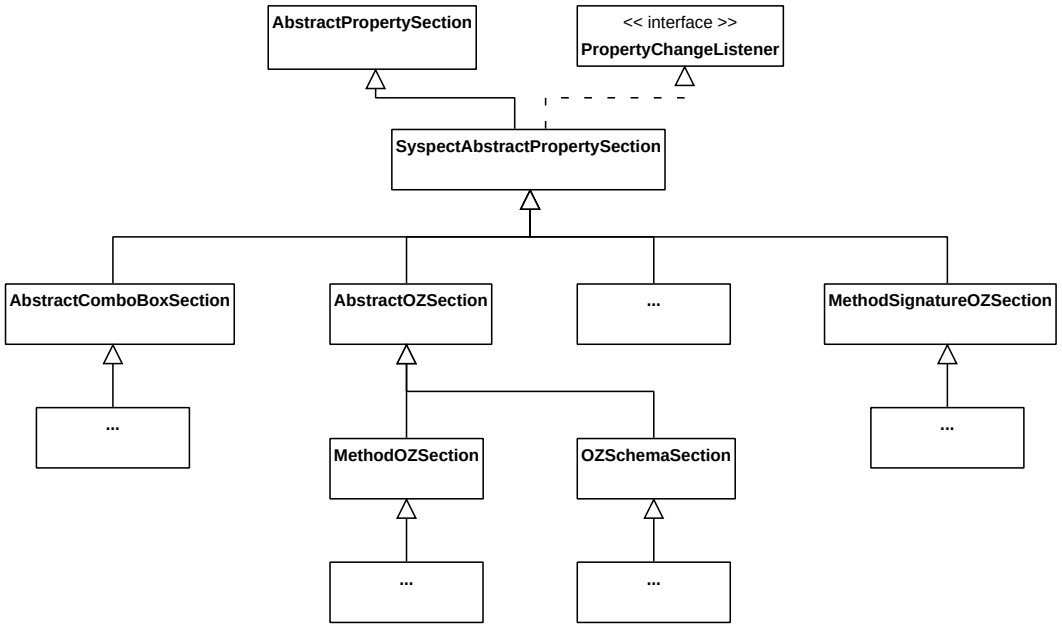


Abbildung 15.32: Section-Klassen

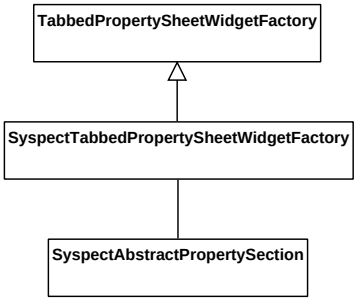


Abbildung 15.33: Widget Factory

`org.eclipse.core.runtime`,
`org.eclipse.gef` und
`org.eclipse.ui`

15.3.13 SyspectCSPOZDCExport Plug-in

Der CSP-OZ-DC-Export stellt zum einen die Funktionalität bereit, unser UML-Modell nach CSP-OZ-DC zu exportieren. Dieses wird beispielweise vom PEA-Export benutzt um den Export vorzubereiten. Desweiteren beinhaltet er auch die Funktionalität die CSP-OZ-DC-Spezifikation nach CSP-OZ-DC-XML, als auch nach LaTeX zu exportieren.
Die Datenstruktur, die eine CSP-OZ-DC-Spezifikation repräsentiert, ist in Abbildung 15.34 zu sehen. Innerhalb dieser Datenstruktur wird der Zugriff auf den Object-Z-Teil einer Klasse über das Interface `ClassInterface` realisiert, da die Übersetzung

aus der UML-Datenstruktur in Object-Z für `IData`- und `ICapsule`-Klassen analog verläuft. Eine CSP-OZ-DC-Klasse enthält zusätzlich noch ein `ProcessEquationSystem`, welches die CSP-Prozesse darstellt, und eine Menge an `DCCounterTrace`s, welche die DC-Gegenbeispiel-Formeln enthalten.
Instanzen dieser Datenstruktur werden von der Klasse `UML2CSPOZDCJConverter` erzeugt, wobei die eigentliche Übersetzung auf mehrere Hilfsklassen aufgeteilt wurde. Zustandsautomaten werden durch `SyspectSM2CSPConverter` in CSP-Prozessysteme übersetzt und mit den durch den `ComponentTable2CSPConverter` erzeugten CSP-Prozessen, die die für die Klasse relevanten Komponentendiagramme darstellen, vereinigt. Die Object-Z-*tagged values* der UML-Klassen werden von der Klasse `UML2ObjectZConverter` in die entsprechenden Object-Z-Elemente übersetzt. Schließlich werden aus den DC-Ausdrücken mittels der Klasse `String2CountertraceConverter` Syntaxbäume erzeugt.
Um aus dieser Datenstruktur CSP-OZ-DC-XML und LaTeX zu erzeugen, besitzen die entsprechenden Klassen `toLaTeX`- und `toXML`-Methoden, die das gewünschte Format in eine übergebene `Writer`-Instanz schreiben.

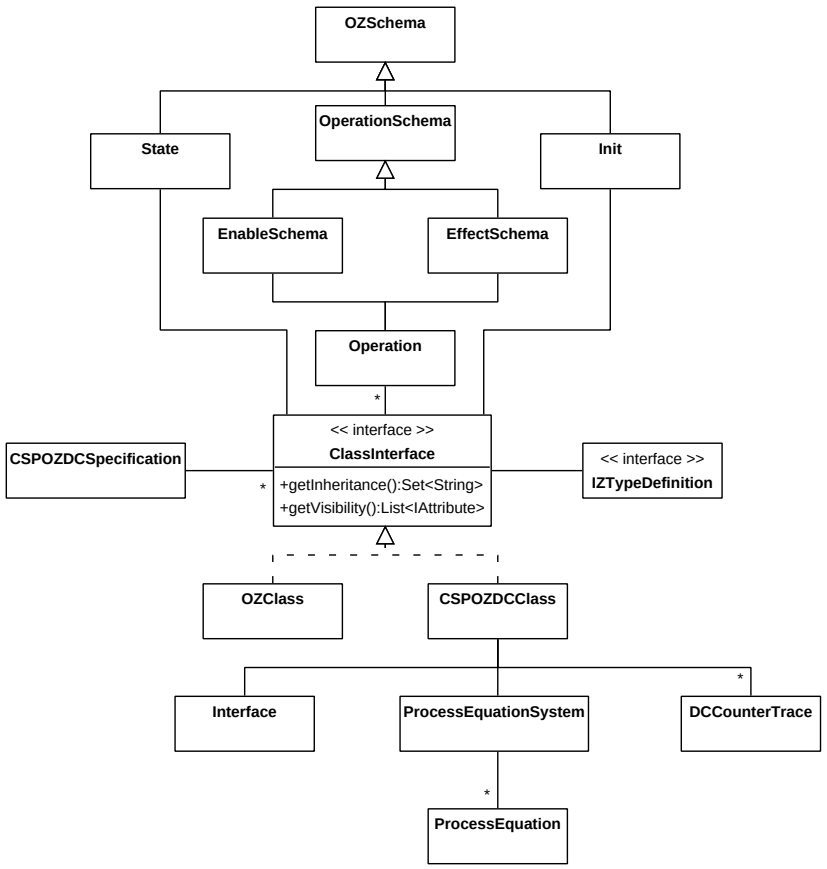


Abbildung 15.34: CSP-OZ-DC-Spezifikation

Schnittstellen

Dieses Plug-in erweitert die zwei Extension Points `org.eclipse.ui.actionSets` und `org.eclipse.ui.popupMenus`, sowie den Extension Point `org.eclipse.help.toc` (siehe Abschnitt 15.2.2, S. 154).

Desweiteren hat das Plug-in Abhängigkeiten zu folgenden Plug-ins:
`de.syspect.componenttable`,
`de.syspect.core`,
`de.syspect.ozdcutil`,
`de.syspect.persistence`,
`de.syspect.utils`,
`org.eclipse.core.runtime` und
`org.eclipse.ui.views` `org.eclipse.ui`

Übersetzung der Komponentendiagramme

Die Übersetzung der Komponentendiagramme geschieht in 6 Phasen:

1. Tabelle erstellen (Aufsammeln der bestehenden Verbindundungen)
2. Zusammenfassen (Erkennen gemeinsamer Verbindungen)
3. Konfliktüberprüfung (Erkennen überlappender Verbindungen)
4. Zweites Zusammenfassen (Eliminieren enthaltener Verbindungen)
5. Priorisierung
6. Übersetzung nach CSP

Im folgenden werden diese Phasen genauer beschrieben.

Tabelle erstellen Zuerst wird eine leere Tabelle angelegt, die Spalten der Tabelle werden beschriftet mit (AssemblyConnector, Interface, Provide, Require). Für jeden AssemblyConnector (AC) erstellen wir eine Zeile in dieser Tabelle, wobei wir in die Interface-Spalte das zugehörige Interface, in die Provide-Spalte die bereitstellende(n) Komponente(n) und in die Require-Spalte die benötigende(n) Komponente(n) schreiben. Hierbei muss darauf geachtet werden, dass die Komponenten-Instanzen unterschieden werden. Dies kann durch Indizierung geschehen.
Eine Zeile in der Tabelle beschreibt jeweils eine Verbindung von Komponenten durch einen AssemblyConnector A. Für die Informationen in den Spalten definieren wird die folgenden Abkürzungen:

$$
\begin{aligned}
int(A) &:= \text{das Interface zum AssemblyConnector A} \\
r(A) &::= \text{alle benötigenden Komponenten, am AssemblyConnector A.} \\
p(A) &::= \text{alle bereitstellenden Komponenten, am AssemblyConnector A.} \\
comp(A) &::= r(A) \cup p(A)
\end{aligned}
$$

Im Folgenden bezeichne $(A_i)_{i=1..n}$ die so erstellte Tabelle.

Zusammenfassen Nun werden paarweise alle Zeilen der Tabelle verglichen, und sofern die Interfaces identisch sind sowie die Einträge in der Requirespalte oder die Einträge in der Providespalte, so wird eine der Zeilen gelöscht.
Dieses Verfahren wird in zwei Durchläufen durchgeführt. Zunächst werden von jeweils zwei Zeilen (A_i, A_j mit $i, j \in 1..n \wedge i \neq j \wedge A_i, A_j \notin \mathit{Deleted}$) mit identischem Interface ($\mathit{int}(A_i) = \mathit{int}(A_j)$) die Requirespalte verglichen. Sind sie identische ($r(A_i) = r(A_j)$) so handelt es sich um eine gemeinsame Verbindung. In dem Fall werden die Komponenten der Providespalte vereinigt. Eine der beiden Spalten wird gelöscht ($\mathit{Deleted}' = \mathit{Deleted} \cup \{A_j\}$) und die jeweils andere erhält die neue Provides-Information ($p(A_i)' = p(A_i) \cup p(A_j)$).
Im zweiten Durchlauf tauschen Requires- und Providespalte die Rollen. D.h. bei gleichem Interface ($\mathit{int}(A_i) = \mathit{int}(A_j)$) werden die Providespalten vergleichen und bei Gleichheit ($p(A_i) = p(A_j)$) die Requires-Information vereinigt ($r(A_i)' = r(A_i) \cup r(A_j)$). Durch dieses Verfahren werden AssembleyConnectoren entdeckt, die semantisch äquivalent durch einen einzigen Connector ausgedrückt werden könnten. Die zusammengefasste Tabelle $(A_i)'_{i=1..n'} = (A_i)_{i=1..n} \setminus \mathit{Deleted}$ beschreibt diese vereinfachten Verbindungen.

Konfliktüberprüfung In der Konfliktüberprüfung wird nun sichergestellt, dass die verbleibenden Verbindungen entweder konfliktfrei durch andere überdeckt werden, oder aber dass sie völlig unabhängig voneinander sind. Ist beides nicht der Fall – die Komponentenverbindungen liegen also „schief“ zueinander – dann kann keine Übersetzung nach CSP erfolgen.
Zunächst definieren wir eine Funktion, die uns die gemeinsamen Komponenten von zwei Verbindungen A und B heraussucht:

$$\mathit{share}(A, B) := \mathit{comp}(A) \cap \mathit{comp}(B)$$

Komponenten können nur „schief“ zueinander liegen, wenn diese Schnittmenge nicht leer ist. In dem Fall gilt es zu überprüfen, ob die gemeinsamen Komponenten verträglich zueinander in beiden Verbindungen benutzt werden. Verträglich können sie zum einen sein, wenn eine Verbindung die gemeinsamen Komponenten überdeckt. Falls Verbindung A eine Überdeckung von B ist so schreiben wir $A \geq B$ mit:

$$\begin{aligned} A \geq B &\equiv \mathit{share}(A, B) \subseteq r(A) \vee \mathit{share}(A, B) \subseteq p(A) \\ A > B &\equiv A \geq B \wedge \neg\, B \geq A \end{aligned}$$

A ist also eine Überdeckung von B, falls die gemeinsamen Komponenten komplett auf der Require- oder Provideseite von A auftauchen. A ist genau dann eine *echte Überdeckung* von B, wenn B dabei keine Überdeckung von A ist. Wir sagen die Verbindungen sind überdeckungskompatibel, genau dann wenn eine der Komponenten die andere echt überdeckt:

$$\mathit{cover}(A, B) \equiv A > B \vee B > A$$

Zum anderen sind die Verbindungen verträglich, wenn (ggf. durch Vertauschung der Rollen bei einer Verbindung) Require und Provide auf den gemeinsamen Komponenten

übereinstimmen:

$$\begin{aligned} \mathit{compatible}(A,B) \quad \equiv \quad & (((r(A) \cap \mathit{share}(A,B)) = (r(B) \cap \mathit{share}(A,B))) \\ & \wedge ((p(A) \cap \mathit{share}(A,B)) = (p(B) \cap \mathit{share}(A,B)))) \\ & \vee (((r(A) \cap \mathit{share}(A,B)) = (p(B) \cap \mathit{share}(A,B))) \\ & \wedge ((p(A) \cap \mathit{share}(A,B)) = (r(B) \cap \mathit{share}(A,B)))) \end{aligned}$$

Alternativ kann dieses auch wie folgt überprüft werden:

$$\begin{aligned} \mathit{compatible}(A,B) \quad \Leftrightarrow \quad & ((r(A) \cap r(B)) \cup (p(A) \cap p(B)) = \varnothing \\ & \vee ((r(A) \cap p(B)) \cup (p(A) \cap r(B)) = \varnothing \end{aligned}$$

Wieder gehen wir paarweise alle Zeilen der Tabelle durch ($\#\mathit{Tests} \leq n(n-1)$). Sei A die erste und B die zweite betrachtete Zeile. Ein Konflikt zwischen den Zeilen besteht genau dann wenn:

$$(\mathit{share}(A,B) \neq \varnothing) \wedge \neg\, \mathit{cover}(A,B) \wedge \neg\, \mathit{compatible}(A,B)$$

In dem Fall sind also die Verbindungen der Komponenten $\mathit{share}(A,B)$ über die Interfaces $\mathit{int}(A) \cup \mathit{int}(B)$ nicht kompatibel miteinander.

Priorisierung Die Priosierung soll sicherstellen, dass Verbindungen, deren Komponenten Teil einer anderen Verbindung sind, favorisiert übersetzt werden. Es werden also quasi Subkomponenten gebildet, die von der kleinsten zur größten übersetzt werden.
Für die Priorisierung zählen wir einfach wie oft eine Verbindung A_i eine andere überdeckt.

$$\mathit{prio}(A_i) := \mid \{A_j \mid A_i \neq A_j \wedge A_i > A_j\} \mid$$

In der Implementierung wird die hier zur Bestimmung der Priorität benutzte Menge für jede Verbindung auch festgehalten, um die Neuberechnung der Priorität nach einem Übersetzungsschritt (s.u.) effizient zu halten.

Zweites Zusammenfassen Wiederum werden paarweise alle Zeilen der Tabelle betrachtet. Hierbei seien die AssemblyConnectoren mit A und B bezeichnet und $\mathit{share}(A,B)$ ist die Menge der gemeinsamen Komponenten von A und B. Die beiden Zeilen werden genau dann zusammengefasst, wenn

$$\mathit{share}(A,B) \neq \varnothing \wedge \mathit{compatible}(A,B) \wedge \mathit{prio}(A) = \mathit{prio}(B),$$

also wenn die beiden Verbindungen gemeinsame Komponenten haben, sich nicht überdecken und kompatibel sind, also die gemeinsamen Komponenten jeweils auf der gleichen Seite der Verbindungen liegen (ggf. nach Vertauschung der Rollen bei einer Verbindung).
Zusammengefasst werden die Zeilen, indem die neue Zeile die Überdeckung der einen Verbindung durch die andere ausdrückt. Hierbei muss sichergestellt werden, dass in der

neuen Zeile die Require- und Providespalten jeweils nur Elemente vereinigen, die nicht schon in der jeweils anderen Spalte auftauchen. Das Prädikat φ soll ausdrücken, ob gemeinsame Komponenten die gleiche Rolle in beiden Verbindungen einnehmen, also auf der gleichen Seite der Verbindung auftauchen:

$$\varphi \equiv (r(A) \cap r(B) \neq \varnothing \vee p(A) \cap p(B) \neq \varnothing)$$

Für die neue Zeile erhalten wir dann die Vereinigung der Interfaces $int'(A) = int(A) \cup int(B)$. Wenn die gemeinsamen Komponenten die gleiche Rolle einnehmen, dann werden Require- und Providespalte direkt vereinigt.

$$\varphi \Leftrightarrow ((r'(A) = r(A) \cup r(B)) \wedge (p'(A) = p(A) \cup p(B)))$$

sonst werden sie mit vertauschten Rollen bei einer der Verbindungen zusammengefasst.

$$\neg\, \varphi \Leftrightarrow ((r'(A) = r(A) \cup p(B)) \wedge (p'(A) = p(A) \cup r(B)))$$

Die ursprünglichen Zeile B wird dann gelöscht. Bei der jeweils nächsten Überprüfung ist zu beachten, dass sich die Prioritäten aller Verbindungen ggf. durch die neue Zeile geändert haben kann.

Übersetzung nach CSP Bei der Übersetzung eines Komponentendiagramms müssen nun alle Komponenten, die in dem Diagramm auftauchen, als Prozesse instantiiert werden.

$$Inst := \{C_i \mid C_i \text{ ist im Diagramm instantiierte Komponente}\}$$

Die Komponenten, die hierbei mit anderen Komponenten verbunden sind, finden wir in der erzeugten Tabelle $(A_i)_{i=1..n}$ wieder:

$$\bigcup_{i=1..n} comp(A_i) \subseteq Inst$$

Wir beginnen mit der Übersetzung der verbundenden Komponenten. Dazu werden die Zeilen der Tabelle nun nach Priorität geordnet abgearbeitet. Begonnen wird mit einer Zeile mit der niedrigster Priorität. Für eine Komponente C, eine Verbindung I und einen Prozess PI definieren wir uns induktiv eine Alphabetfunktion:

$$\begin{aligned}
im(C) &:= \text{alle Methoden von C, die Interfaces zugeordnet sind} \\
im(I) &:= \text{alle Methoden aus Interfaces von } int(I) \\
proc(PI) &:= \text{alle Prozesse (bzw. Komponenten), die in PI vorkommen} \\
\alpha(C) &:= im(C) \\
\alpha(PI) &:= \bigcup_{x \in proc(PI)} \alpha_{comp}(x)
\end{aligned}$$

Weiterhin werden noch einige Hilfsfunktionen benötigt.

$$br(alphabet) := \text{alle Broadcast-Methoden von } alphabet$$
$$nth(A, s, i) := \text{liefert die i-te Komponente aus } r(A) \text{ (für s=r) bzw. } p(A) \text{ (für s=p) der Tabellenzeile mit AssemblyConnector A}$$

Sei nun I der Name des Interfaces der betrachteten Zeile. Dann definieren wir uns zwei Prozesse PI_{r,n_1} und PI_{p,n_2} mit $n_1 =| r(I) |$ bzw. $n_2 =| p(I) |$:

$$PI_{s,1} := nth(I, s, 1) \qquad \text{mit } s \in r, p$$
$$PI_{s,i} := PI_{s,i-1} \underset{br(\alpha(PI_{s,i-1}) \cap \alpha(nth(I,s,i)))}{\parallel} nth(I, s, i)$$

Dann ergibt sich PI wie folgt:

$$PI = PI_{r,n_1} \underset{im(I) \cup br(\alpha(PI_{r,n_1}) \cup \alpha(PI_{r,n_1}))}{\parallel} PI_{p,n_2}$$

Sollte es mehrere Zeilen in der Tabelle geben, die auf das gleiche Interface verweisen, so muss bei der Übersetzung der Zeile das Interface umbenannt werden.

$$PI_i = (PI_{i,r,n_1} \underset{im(I) \cup br(\alpha(PI_{r,n_1}) \cup \alpha(PI_{r,n_1}))}{\parallel} PI_{i,p,n_2})[I/I_i]$$

Nach dem Übersetzen einer Zeile werden in den verbleibenden Verbindungen die gerade instanziierten Komponenten durch die neue (virtuelle) Komponente ersetzt. Hierzu definieren wir wieder eine Funktion *subst*, die in der Menge X alle Vorkommen von Elementen aus Y durch das Element x ersetzt:

$$subst(X, Y, x) := (X \setminus Y) \cup \{| \, X \cap Y \neq \varnothing \bullet x\}$$

Für aus der Übersetzung hervorgegangene Prozesse können wir dann definieren, dass die Komponentenvorkommen $comp(PI)$ durch die neue Komponente PI ersetzt werden:

$$subst_P(X, PI) := subst(X, comp(PI), PI)$$

Damit erhalten wir für die aktualisierte Tabelle:

$$\forall\, i : 1..n \mid A_i \neq I \bullet r'(A_i) = subst_P(r(A_i), PI) \wedge p'(A_i) = subst_P(p(A_i), PI)$$

Auch unsere Menge zu instantiierender Komponenten aktualisieren wir entsprechend:

$$Inst' = subst_P(Inst, PI)$$

Anschließend wird die Zeile I aus der Tabelle entfernt und die Zeilen neu priorisiert. Hierzu kann die aktuelle übersetzte Zeile I aus den Basismengen für die Priorisierung einer jeden übrigen Zeile entfernt werden, um so effizient die neue Priorität zu erhalten.

Nun wird mit aufsteigender Priorität mit der Übersetzung der nächsten Tabellenzeile fortgefahren.
Wurde das letze Interface übersetzt, müssen noch Interfaces zurückbenannt werden und der entstehende Prozess parallel zum Prozess der Statemachine der Komponente erzeugt werden.

$$
\begin{aligned}
comp &:= \Big|\Big|\Big|_{PX \in Inst} PX[\{(p,q) \mid \text{ p ist Instanz von q (und umbenannt)}\}] \\
main &:= main_{SM} \underset{\alpha(main_{SM}) \cap \alpha(comp)}{\|} comp
\end{aligned}
$$

15.3.14 SyspectPEAExport Plug-in

Der PEA Export exportiert aufbauend auf dem CSP-OZ-DC-Export unsere UML-Spezifikation zu Phasen-Event-Automaten. Mithilfe der PEA-Tools kann er darüber hinaus diese auch zu PEA-XML exportieren. Das Plugin besteht primär aus 3 Convertern, jeweils einem für jede Spezifikationskomponente. Die Klasse `pea.modelchecking.PEAJ2XMLConverter` aus den PEA-Tools wird benutzt um die XML-Darstellung der Phasen-Event-Automaten zu erstellen. Bislang ist es nicht möglich den CSP-Rename-Operator zu übersetzen. Daher können Spezifikation, die u.a. aus Komponentendiagrammen erstellt wurden nicht in PEA übersetzt werden.

Schnittstellen

Dieses Plug-in erweitert ebenfalls nur die Extension Points `org.eclipse.ui.actionSets` und `org.eclipse.ui.popupMenus`, sowie `org.eclipse.help.toc` (siehe Abschnitt 15.2.2, S. 154).

Weitere Abhängigkeiten bestehen zu folgenden Plug-ins:
`de.syspect.core`,
`de.syspect.export.cspozdc`,
`de.syspect.ozdcutil`,
`de.syspect.persistence`,
`de.syspect.utils`,
`org.eclipse.core.runtime`,
`org.eclipse.ui.views` und
`PEA_Tool`

15.3.15 SyspectImageExport Plug-in

Der ImageExport nutzt den `EditPartFactory`-Extension-Point des core's um die verschiedenen Diagramme in die Formate BMP, EPS, JPEG und PDF zu exportieren. Der Export geschieht über Frameworkfunktionen von SWT. Wir benutzen hierbei die *Editpart*-Factories der Editoren, um die Diagramme "in" die Bilder zu zeichnen. Für den EPS Export im speziellen benutzen wir darüber hinaus Funktionen von GMF. Im Moment sind wir hierbei auf eine ältere Version von GMF angewiesen, da bei der aktuellen, die verwendeten Funktionen nicht mehr sichtbar sind.

Schnittstellen

Dieses Plug-in erweitert ebenfalls die Extension Points `org.eclipse.ui.actionSets`, `org.eclipse.ui.popupMenus`,und `org.eclipse.help.toc` (siehe Abschnitt 15.2.2, S. 154).

Weitere Abhängigkeiten:
`de.syspect.core`,
`de.syspect.navigator`,
`de.syspect.persistence`,
`de.syspect.utils`,
`org.eclipse.core.runtime`,
`org.eclipse.gef`,
`org.eclipse.gmf.runtime.draw2d.ui.render` und
`org.eclipse.ui.views`

15.3.16 SyspectComponentTable Plug-in

Das ComponentTable Plugin implementiert den ersten Teil der Komponenten-Diagramm-Übersetzung (siehe Abschnitt 15.3.13, S. 196). Es wird benutzt um sowohl dem ModelConsistencyChecker zu ermöglichen Konflikte zu erkennen, als auch dem Export auf die entsprechenden, bereits aufgebauten Datenstrukturen zuzugreifen.

Schnittstellen

Dieses Plug-in hat Abhängigkeiten zu folgenden drei Plug-ins:
`de.syspect.core`,
`org.eclipse.core.runtime` und
`org.eclipse.ui.views`

15.3.17 Richtlinien für den Quellcode

Die Quellcode-Richtlinien von *Syspect* dienen der Einheitlichkeit und Lesbarkeit des Quellcodes und somit der Wiederverwendbarkeit und Wartbarkeit des Produktes.
Grundlage für unsere Quellcode-Richtlinien sollen die Java Code Conventions [Mic97] sein.

Es folgen einige Erweiterungen bzw. Hervorhebungen besonders wichtiger Regeln:

- Sprache: Bezeichnernamen und Kommentare des Quellcodes sind in englisch zu verfassen.
- Bezeichnernamen sollten selbstverständlich selbsterklärend sein und keine kryptischen Abkürzungen oder ähnliches enthalten.
- Es soll unterschieden werden zwischen Dokumentationskommentaren, die beschreiben, was eine Methode tut und Implementierungskommentaren, die bei

besonders komplexen Quellcode eingesetzt werden, welcher Gefahr läuft, nicht selbsterklärend zu sein. Implementierungskommentare beschreiben, wie eine Methode arbeitet.

- Für Dokumentationskommentare ist javadoc einzusetzen. Jede Klasse und jede Methode sollte mit javadoc kommentiert sein! Dokumentationskommentare sollten insbesondere zwei Ziele verfolgen:

  1. Sie sollen es einem Programmierer ermöglichen, die Klasse als Blackbox zu benutzen, ohne den Quellcode zu kennen.
  2. Sie sollen es einem Testprogrammierer ermöglichen, sinnvolle Testfälle abzuleiten.

- Bevor man Implementierungskommentare verwendet, sollte versucht werden, den Code so umzuschreiben, dass die Implementierungskommentare nicht mehr nötig sind. Durch selbsterklärende Bezeichnernamen, kurze Methoden (zur Orientierung: nur in Ausnahmefällen über 30 Zeilen!) und Implementierungskommentare muss ein Programmierer, der eine Methode nicht selbst verfasst hat, möglichst schnell und dabei präzise erfassen können, wie sie arbeitet.

- Wichtige Entscheidungen sollten ebenfalls dokumentiert werden. Beispiel: Wenn ich drei Möglichkeiten sehe, die Funktionalität einer Methode zu realisieren und mich aus bestimmten Gründen für eine Möglichkeit entscheide bzw. aus bestimmten Gründen gegen die beiden anderen, dann sollte dieser Vorgang dokumentiert werden. Eine Anmerkungen dazu: Was „wichtig“ in diesem Fall bedeutet, ist subjektiv und sollte zunächst vom Programmierer entschieden werden. Allerdings sollte bei Review-Phasen konstruktive Kritik an der Dokumentation geübt werden, um diese im Zweifelsfall zu präzisieren.

- Jeder Klasse soll ein Verantwortlicher zugeordnet werden. Dieser wird zu Beginn als Autor erwähnt und ist im Normalfall der, der die erste Version der Klasse eincheckt. Für den Fall, dass ein anderer an der Klasse etwas ändern muss, hat er zu markieren, dass die Änderung nicht vom Autor stammt und er hat den Autor zum nächstmöglichen Zeitpunkt zu kontaktieren und die Änderung mit ihm abzusprechen. Dadurch erhalten wir für jeden Schnipsel Code einen klaren Ansprechpartner. Natürlich kann im Ausnahmefall die Verantwortung für eine Klasse mal getauscht werden bzw. kommissarisch und zeitlich begrenzt von einem anderen geführt werden (z.B. wenn einer Urlaub hat).

  Nachtrag: Durch die große Anzahl der Klassen in *Syspect*, der häufig eingesetzten Praxis, dass einige wenige viele Klassenrümpfe erstellen, die dann nach und nach von anderen umgearbeitet werden und bestimmt auch zu wenig Disziplin wurde diese Richtlinie kaum eingehalten.

- Weitere Richtlinien:

  – Für den Fall, dass for-Schleifen nicht in der neuen 5er-Schreibweise verfasst sind, sondern mit Indexvariablen, sollte hier immer „i“ verwendet werden. Bei verschachtelten for-Schleifen dann weiter „ j“, „k“, usw.

  - Einschränkung der Methodenlänge: Die Länge einer Methode sollte nur im begründeten Ausnahmefall eine Länge von 30 Zeilen (oder einem ähnlichen Wert) überschreiten.
  - In keinem Fall sollten Strings fest codiert im Code erscheinen, sondern es sollte einerseits eine Lokalisierung stattfinden und andererseits Enums bzw. Konstanten verwendet werden.

- Abschliessend lässt sich sagen, dass Lesbarkeit definitiv vor Kürze geht.

Klassennamen Abgesehen von den Namenskonventionen im Model (siehe Abschnitt 15.3.6, S. 167) gab es folgende Namenskonventionen:

- EditPolicies erhalten das Suffix „EditPolicy“ (z.B. MethodEditPolicy)
- Suffix „Figure“ für Figures (z.B. MethodFigure)
- Suffix „Command“ für Commands (z.B. DeleteCommand)

Teil III

Zusammenfassende Bewertung und Ausblick

Kapitel 16

Zusammenfassende Bewertung

In diesem Abschnitt soll die erstellte *Syspect*-Applikation und seine Entwicklung zusammengefasst beschrieben und die Vor- und Nachteile dargelegt werden.

Über eine Produktanalyse sollte die Plattform für das *Syspect*-Projekt gefunden werden. Diese Analyse beruhte zu meist auf Kenntnissen der Beteiligten aus den Seminaren. Aufgrund der offensichtlichen Vorteile der *Eclipse*-Plattform entschied sich die Projektgruppe dafür, diese als Plattform für die *Syspect*-Applikation zu verwenden. Ausschlaggeben war dafür vor allem das Eclipse-Plug-in-Konzept, das es ermöglicht, nicht nur die eigene Applikation auf einem hohen Level der Erweiterbarkeit, Wartbarkeit und Wiederverwendung zu halten, sondern auch die Verwendung von schon bestehenden Plug-ins. Die Grundfunktionalitäten einer GUI wurden somit vom dem Eclipse-Framework und die eines grafischen Editors von dem GEF-Plugin zu Verfügung gestellt. Zudem ist Eclipse Open Source und hat eine große Community bzw. eine daraus resultierende gute Dokumentation und ist in der Anwendung getestet.
Daneben wird noch weitere Software (z.B. *Community Z Tools*) als Komponenten in das Projekt mit eingebettet. Zusätzlich zur Integration in *Eclipse* besteht die Möglichkeit das Projekt als *Rich Client Application* zu realisieren. Somit entsteht ein eigenständiges Produkt auf Basis der *Eclipse*-Plattform.

Über das Plug-in-Konzept können komponentenbasierten Systeme entwickelt werden. Dies ermöglicht eine einfache Skalierung von *Syspect*, obwohl es Grundkomponenten gibt, wie das Syspect Core Plug-in, die in jedem Fall vorhanden sein müssen. Einstellungen, wie Abhängigkeiten oder Extension Points, der einzelnen Plug-ins können über die Plug-in Editor Tools komfortabel im Menü oder direkt in der plugin.xml editiert werden.

Beim Feinentwurf werden die einzelnen Komponenten definiert, d.h. die Plug-in-Struktur für *Syspect* erstellt. Beispielhaft dargestellt sei die Ausgliederung der Views und Editoren in einzelne Plug-ins. Diese umfassen die Diagramm-Editoren sowie den Navigator. Die Editoren besitzen Abhängigkeiten zum Navigator und zum SyspectDiagramEditorBundle-Plug-in. Dieses stellt allgemeine Funktionalitäten für die Editoren zur Verfügung. Die Editoren benutzen zur Darstellung von Objekten *GEF*. Dieses bedient sich der MVC-Architektur, so dass eine strukturierte Klassen-

hierarchie innerhalb der Editor-Plug-ins garantiert wird. Diese Architektur liegt auch den meisten anderen Komponenten zu Grunde. Folglich ist die Paketstruktur der Plug-ins einheitlich. Als Stammpaket wurde entsprechend der Konvention `de.syspect` gewählt.

Für eine Bewertung sei hier auf die Strukturierung des Models eingegangen. In einer Redesign-Phase wurde ein Model, das für die Lösung komplexer Probleme nicht mehr ausreichte, durch ein neu entworfenes Model ersetzt. Das neue Model basiert auf einer Interface-Struktur, die den Austausch der Modelimplementierung erlaubt. Die Interface-Struktur befindet sich im Syspect Core Plug-in und das konkrete Model wurde in das SyspectModel-Plug-in ausgelagert. Nach der Umstellung des Models wurde das gesamte Projekt auf dieses angepasst. Komplexe Problemstellungen wurden vereinfacht. Im Gegensatz dazu mussten bereits gelöste Probleme neu diskutiert werden.

Abschließend soll die Test-Methodik bzw. die Evaluierung erörtert werden. Das Testen des Systems sollte durch JUnit-Tests erreicht werden. Nach der Redesign-Phase, die eine komplexe Plug-in-Struktur hervorbrachte, wurde aus zeitlichen Gründen auf den Aufbau einer Testsuite verzichtet. Im Kapitel Ausblick soll dieser Aufbau näher erläutert werden. Für die Evaluierung des Softwareproduktes wurde eine Fallstudie durchgeführt. Diese befasst sich mit der Erstellung eines Mail User Agent (MUA) [Sta06]. Durch diesen Workflow wurden viele Funktionen von *Syspect* getestet. Da sich die angegebene Fallstudie nur mit CSP-OZ beschäftigt, war es damit nicht möglich, Funktionen zu testen, die im Zusammenhang mit dem Duration Calculus stehen. Diese mussten anhand anderer Beispiele getestet werden.

16.1 Fallstudie *E-Mail-Client*

Die Fallstudie *E-Mail-Client* [Sta06] behandelt die Modellierung eines Softwaresystems zum Lesen elektronischer Post mittels des UML-Profils für CSP-OZ-DC, das in Abschnitt 9, S. 77 vorgestellt wurde. Das Softwaresystem soll E-Mails anzeigen können, die im Anhang Text und Bildern enthalten können. Der Nutzer kann E-Mails auswählen und deren Anhänge anzeigen lassen. Die drei Komponenten des Systems werden als *Mail User Agent*, *TextViewer* und *ImageViewer* bezeichnet. Der *Mail User Agent* kontrolliert die Kommunikation mit dem Nutzer, während die beiden *Viewer* die Anhänge der E-Mails darstellen können.
Da sich das UML-Profil für CSP-OZ-DC im Laufe des Projekts geändert hat, wie in Abschnitt 14.2, S. 143 beschrieben, musste auch die Modellierung in Syspect angepasst werden. Die Unterschiede und Besonderheiten, die beachtet werden müssen, werden anhand des Zustandsautomaten des *Mail User Agent*, dem Klassendiagramm, in dem die Schnittstellen des *Mail User Agent* definiert werden und dem Komponentendiagramm, das das System spezifiziert dargestellt.

Kapitel 17

Ausblick

In diesem abschließenden Kapitel sollen Features und Konzepte vorgestellt werden, die im Laufe unseres Projektes zwar diskutiert wurden, aber nicht in die endgültige Version unserer Software aufgenommen wurden. Wenn es dafür andere Gründe gab als die Knappheit der zur Verfügung stehenden Zeit, so sind diese angegeben. Ebenso werden konkrete Schüsselstellen im Quellcode genannt, falls solche für ein Feature gibt.

17.1 EMF

Die Idee des *Eclipse Modeling Framework* ist, dass sich ein Datenmodell, ob es nun in Java, XML oder UML notiert ist, auf gemeinsame Konzepte zurückführen läßt. Das theoretische Model ist das MOF von der OMG (siehe Kapitel 6, S. 49 bzw. [MOF06]). Existiert eine Darstellung eines Datenmodells in Form von Java-Interfaces, läßt sich mit Hilfe von EMF daraus eine Darstellung als UML-Diagramm oder als XML-Schema erzeugen (vgl. Abbildung 17.1). Unabhängig davon, welche Darstellung benutzt wird, um das EMF-Modell zu erzeugen, ermöglicht es EMF, daraus die anderen Darstellungen und sogar die zugehörigen Implementierungsklassen zu generieren. Das zentrale Modell, das benutzt wird, um Modelle in EMF selbst zu repräsentieren, nennt sich *ECore* und läßt sich am besten als kleine vereinfachte Menge der gesamten UML beschreiben.

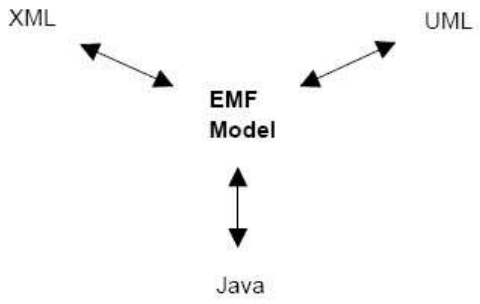


Abbildung 17.1: EMF vereint Java, UML und XML

Zusätzlich zu der bisher beschriebenen Funktionalität kann EMF ein *ECore*-Modell auch als XMI serialisieren. Das wäre für *Syspect* vor allem deswegen von Bedeutung, da fast jedes UML-Werkzeug sein eigenes persistentes UML-Format benutzt. XMI ist als Standard zur Übertragung von UML-Daten vorgesehen.
Es wird jedoch an dieser Stelle darauf hingewiesen, dass bisher kaum ein UML-Modellierungswerkzeug in der Lage ist, das XMI-Format eines anderen Werkzeugs vollständig und korrekt zu lesen (vgl. [RHQ+05], S. 35f). Es wäre also im abschätzbaren Aufwand möglich, eine XMI-Schnittstelle zu realisieren, aber ob sie sinnvoll nutzbar ist, ist zumindest als fraglich zu bezeichnen.

17.2 UML2

Das UML2-Plug-in für *Eclipse* ist eine Implementierung der UML 2.0 in Java. Das Plug-in basiert auf EMF.
Innerhalb des UML2-Plug-ins sind Klassen für die für *Syspect* erforderlichen in Klassen-, Komponenten- und Zustandsdiagrammen enthaltenen Elemente vorhanden. Die Verbindung dieser Elemente geschieht über Klassen, die Assoziationen und Transitionen modellieren. Das Plug-in unterstützt weiterhin die Erweiterung eines Modells über Profile und Stereotypen, was essentiell für die Anwendung innerhalb *Syspects* ist. Ein innerhalb des Plug-ins entwickeltes Modell lässt sich direkt in das Austauschformat XMI exportieren.
Zum Erzeugen der einzelnen Elemente werden sowohl die Klasse `UML2Factory`, welche das Design-Pattern *Singleton* [GHJ95] implementiert, als auch die Repräsentationsklassen der konkreten UML 2.0-Elemente, welche hierfür *Factory-Pattern* [GHJ95] bereitstellen, benutzt.
Da die Entwickler großen Wert auf die Konformität des Plug-ins zur UML 2.0-Spezifikation legen, ist es unwahrscheinlich, dass sich bei der Weiterentwicklung wesentliche Schnittstellen und Funktionalitäten ändern.
Das UML2-Plug-in bietet sich auf den ersten Blick als eine gute Datenhaltung für Diagramme an, da die Diagramme UML2 konform sein sollen. Allerdings war die Einarbeitung in das UML2-Plug-in und die Integration in *Syspect* so aufwendig, dass schließlich die fehlende Funktionalität einer *Undo*- und *Redo*-Funktion in Verbindung mit GEF ausschlaggebend war, eine eigenständige Datenhaltung zu entwerfen. Da das UML2-Plug-in – aufbauend auf dem EMF-Plug-in – aber genau die XMI-Export-Funktion zur Verfügung stellt, die *Syspect* benötigen würde, bleibt das UML2-Plug-in als Exportkomponente interessant. Eine Alternative ist die Datenhaltung von Argo-UML, welches jedoch noch problematischer dadurch ist, dass es nur die UML 1.5-Spezifikation implementiert.

17.3 Druckfunktion

Momentan existiert keine funktionsfähige Druckfunktion, um die erzeugten Diagramme direkt aus *Syspect* heraus auf einem Drucker auszugeben. In einer früheren Entwicklungsphase des Programms wurde die Druckfunktion bereits implementiert, besaß

jedoch die Einschränkung, dass es nur auf einer Microsoft Windows Plattform einwandfrei lauffähig war. Aufgrunddessen, sowie dem Grund, dass die Kunden die erzeugten Diagramme hauptsächlich in Dokumente einbetten nöchten, um Sachverhalte zusätzlich dokumentieren zu können, wurde diese Funktion vorerst nicht in die vorliegende Version des Programms aufgenommen, da ein Export des Diagramms in PDF und verschiedene Bildformate möglich ist.
Im Folgenden wird eine Übersicht darüber gegeben, welche Funktionalitäten die Druckfunktion haben könnte:

- Druckfunktion aufrufbar über Kontextmenü im Navigator durch selektieren einzelner oder mehrerer Diagramme, oder durch einen Menüeintrag in der Menüleiste.
- Selektieren des Ausgabegeräts.
- Selektieren der Ausgabeseiten: alle, von/bis.
- Vorschau des Ausdrucks mit folgender Funktionlität.
  - Iterieren über sämtliche Seiten des Ausdrucks.
  - Zoom-Funktionen: *fit to page*, *fit to width*, *fit to height*

17.4 Notizzettelkommentare

Mit „Notizzettelkommentaren“ sind hier die für UML-Modellierungswerkzeuge typischen Kommentarflächen gemeint, die direkt in Diagramme eingezeichnet und mit gestrichelten Linien mit z.B. Klassen verbunden werden können.
In der Sprache von *Syspect* gesprochen sind dies *Viewable*-Kommentare im Gegensatz zu den bereits vorhandenen *Model*-Kommentaren: Wenn in der gegenwärtigen Version von *Syspect* eine Klasse im *Tabbed Property View* kommentiert wird, so gilt dieser Kommentar für alle Vorkommen dieser Klasse in verschiedenen Diagrammen, es handelt sich also um Kommentare in Bezug auf das *Model*. Mit den Notizzettelkommentaren hingegen wäre es möglich, die verschiedenen Kontexte, in denen eine Klasse in mehreren Diagrammen gezeigt wird, separat voneinander zu kommentieren.

Voraussichtlich ist dieses Feature mit den bereits angewandten Möglichkeiten von GEF, Eclipse und unseren eigenen Bibliotheken ohne besondere Schwierigkeit zu lösen.

17.5 TypeChecking

Im Moment wird im Properties-Plugin Z nur auf syntaktische Fehler geprüft, Typfehler werden nicht gefunden. Schlüsselstellen, die angepasst werden müssten, um diese Funktionalität zu erreichen, sind mit entsprechenden Kommentaren versehen und sind in folgenden Klassen zu finden:

- `MethodParameterWizardPage`

- `OZEditorListener`
- `OZPropertiesChangeKeyListener`
- `ZTypeDefinitionsEditorListener`
- `ZTypeDefinitionsPropertiesChangeKeyListener`
- `ZTypeEditorListener`
- `ZTypePropertiesChangeKeyListener`
- `de.syspect.model.core.CapsuleImpl`

17.6 Verschieben von Klassen

Wenn eine Klasse verschoben werden soll und es ist eins ihrer Kinder (also Attribute oder Methoden) selektiert, so ist dies nur schwer möglich. Eine Möglichkeit ist, die Klasse am Titel anzufassen. Fast man stattdessen eins ihrer Kinder an, so wird zwar angedeutet, dass die Klasse verschoben werden soll, jedoch erscheint immer der Verbots-Cursor. Um dies zu lösen, müssten der `DelegateDragTracker` angepasst werden, damit auch solches Bewegen möglich ist. Das Selektieren von Attributen und Methoden(signaturen) kann nicht ausgeschaltet werden, da sonst der Zugriff auf die Properties eben jener nur sehr umständlich möglich wäre.

17.7 Z-Text-Widget

die getter und setter haben echte/korrekte unicode-Strings, intern benutzt es aber Strings mit nur 16-bit Zeichen - und dann kann es auch die unsichbaren Zeichen sichtbar machen...

17.8 AssemblyConnector ConnectionAnchor

Der *AssemblyConnector* ist ein Element, dass die Elemente bereitgestellten und benötigten Schnittstellen im Komponentendiagramm vereint. Da diese jeweils durch eine Linie mit einer Komponente verbunden sind, ist auch der *AssemblyConnector* mit dieser Linie verbunden. Allerdings wird in der Logik des *AssemblyConnectors* nicht unterschieden, zu welcher Schnittstelle eine Linie gehört. D.h. eine Linie die zu einer bereitgestellten Schnittstelle gehört wird nicht an den Teil der *AssemblyConnector*-Figure gesetzt, die erstere repräsentiert. Das gleiche gilt auch für die Linie der benötigten Schnittstelle.

Abbildung **??** veranschaulicht dieses Problem: Bei der Erstellung der Komponente *Component_0* wurde angegeben, sie benötige die Schnittstelle. Trotzdem ist es – durch

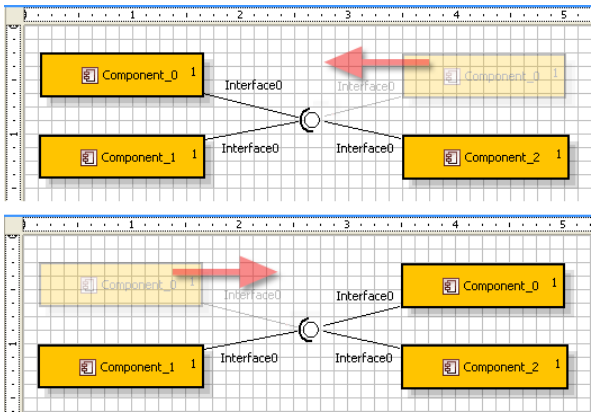


Abbildung 17.2: Assembly

einfaches Verschieben im Diagrammeditor, ohne Änderung der dahinter stehenden Modelldaten – möglich, es so aussehen zu lassen, als stelle sie die Schnittstelle zur Verfügung.
Ein Lösungsansatz wäre, einen *ConnectionAnchor* für den *AssemblyConnector* zu schreiben der diese Eigenschaften erfüllt. Ansatzpunkt für diese Erweiterung wäre die Klasse `AssemblyConnectorEditPart`.

17.9 Übersetzung nach PEA

Da im Moment keine Übersetzung für den CSP-Rename-Operator nach PEA existiert, können Spezifikationen, die Komponentdiagramme enthalten, nicht übersetzt werden. Hier wäre es nötig, eine Möglichkeit zu finden, wie sich der Rename-Operator in PEA ausdrücken lässt und diese im `CSP2PEAConverter` zu implementieren.

17.10 Undo im OZ-Editor

Im OZ-Editor sind derzeit keine *Rückgängig/Wiederherstellen*- Funktionen verfügbar. Da unser bereits implementiertes *Undo/Redo* nicht darauf ausgelegt ist, müsste hier ein eigenes System entwickelt werden. Ansatzpunkt ist die Klasse `OZEditor`.

17.11 Optimierung der Nutzung von javacc

Zur Erstellung des `DC2XMLConverter` wird *javacc* in Verbindung mit dem *Visitor*-Pattern eingesetzt. D.h. vom Entwickler muss nur ein Interface implementiert werden, dass Methoden für den Umgang mit jedem einzelnen Knoten vorschreibt. Die Navigation durch den Parse-Baum übernimmt generierter Code.
Das Problem ist aber, dass die Superklasse `SimpleNode` aller Knotenklassen nicht als `abstract` generiert wird, und somit besagtes Interface auch für diese Klasse eine

`visit`-Methode vorschreibt, die implementiert werden muss.

Da keine Situation möglich erscheint, dass diese Methode tatsächlich aufgerufen wird, ist es zunächst nur ein ästhetisches Problem, diese Methode überhaupt haben zu müssen und in ihr einfach `null` zurückzugeben. Da jedoch nicht abschätzbar ist, wie *Syspect* noch weiterentwickelt wird und durch mögliche Nebeneffekte hier doch weitergehende Probleme auftreten können, wäre eine Beseitigung dieser Schwachstelle wünschenswert.

17.12 Clipped Connections

Clipped Connections sind abgeschnittene Verbindungen in Diagrammen. Diese sind beispielsweise gewünscht, wenn ein Zustand in einer Region außerhalb des sichtbaren Bereichs liegt und eine Transition zu ihm führt. Momentan wird der Zustand selbst dann entsprechend „abgeschnitten“, die Transition hingegen nicht.

Mit der aktuell verfügbaren Version von GEF sind *clipped connections* leider nicht möglich, Ansatzpunkt für diese Funktionalität wäre die Klasse `ContainerFigure`.

17.13 *Drag and Drop* zwischen Projekten

In der aktuellen Version von *Syspect* ist *Drag and Drop* nur innerhalb von Projekten möglich. Um es projektübergreifend zu realisieren, wäre beispielsweise die Implementierung eines *custom transfer* denkbar. Ansatzpunkt für diese Funktionalität ist die Klasse `de.syspect.navigator.View`.

17.14 Effizienteres Event-System

In einer zukünftigen Version von *Syspect* könnte unser *Event*-System (siehe Abschnitt 15.3.6, S. 169) weiter verbessert werden.
Ansatzpunkt wäre unter anderem die Klasse `de.syspect.navigator.ContentProvider`.

17.15 Verschieben von Regionen innerhalb von Zuständen

Momentan ist es nicht möglich, Regionen innerhalb von Zuständen in Zustandsdiagrammen zu verschieben. Um dieses Feature zu implementieren, muss die Klasse `StateFlowLayoutEditPolicy` erweitert werden.

17.16 Definition der Methodentypen in der Konvertierung zu PEA

In `CSP2PEAConverter` for future work: insert definition of the method types?

17.17 Performanz bei der Bendpoint-Berechnung

In `BendpointEditPolicy` for future work: optimize so that handles aren't added constantly.

17.18 Test

Wie bereits in vorangehenden Kapitel erwähnt wird, konnte *Syspect* auf Grund des Model Redesigns kaum getestet werden. Das Testen besteht bisher aus dem Durchführen der Fallstudie MUA.
Testrichtlinien nach denen Tests geschrieben werden sollen, wurden jedoch bereits vorher festgelegt und sollen hier erläutert werden.

17.18.1 Richtlinien für Tests

Diese Richtlinien beziehen sich auf Modultests mit JUnit. Weitere Teststufen, wie z.B. Integrationstests oder Systemtests werden bisher nicht behandelt. Ebenfalls treten auch keine Richtlininen bezüglich GUI-Tests auf.

Allgemeines

- Die Ordnerstruktur der Testklassen wird parallel zur eigentlichen Programmstruktur angelegt.
- Die Testklassen sollten relativ parallel zum eigentlichen Quellcode erstellt werden, d.h. sie müssen nicht vor der Implementierung und auch nicht parallel erzeugt werden. Spätestens bei Abschluss eines Entwicklungsabschnitts (nach Meinung des entsprechenden Entwicklers) sollten vor dem weiteren Vorgehen die nötigen Tests erstellt werden. Diese sollten natürlich funktionsfähig sein, bevor weitergearbeitet wird.
- Testklassen bzw. die Methoden sind entsprechende der Vorgaben der Quellcode-Richtlinien zu erstellen und zu kommentieren. Des Weiteren sollten die Namenskonventionen von JUnit übernommen werden.

Modultests mit JUnit

- Testfälle werden für Methoden aufgestellt, die folgende Kriterien erfüllen:
  - sie sollten eine Parameterliste und einen Rückgabewert haben
  - die Methode sollte min. eigene einfache Operationen (z.B. mathematische Op. oder Op. auf Strings) haben
  - weiter sind Methoden, die Rückgabewerte von Bibliotheksfunktionen liefern bzw. deren Rückgabewerte wieder Eingabewerte für weitere Funktionen sind, in jedem Fall zu testen (also Methoden, die diverse Bibliothksmethoden intern benutzen)

- Um im Programmcode Methoden nicht zu verwenden, die nur für Testzwecke benötigt werden, sollten die assert-Methoden auf primitive Datentypen angewandt werden. Dies betrifft insbesondere die equals-Methode, die man sonst immer überschreiben müsste.
- Testfälle sollten, wenn möglich, um Testobjekte gruppiert werden.

Testsuiten

- Jeder Entwickler sollte seine aufgestellten Testklassen zu einer Testsuite zusammenfassen, bzw. mehrere Testsuiten aufstellen.
- Zentral werden die Testsuiten von jedem einzelnen zu einer gesamten Testsuite zusammengefasst. Neu hinzugekommene Testklassen bzw. -suiten werden dieser gesamten Testsuite selbstständig hinzugefügt.

Anhang

Phasen-Event-Automaten

Phasen-Event-Automaten (PEA) [Hoe05] liefern einen Formalismus, mit dem es möglich ist, eine kompositionelle Semantik für CSP-OZ-DC zu formulieren. Ein PEA ist ein 8-Tupel $(P, V, A, C, E, s, I, Init)$. Die einzelnen Komponenten sind:

- P: Zustandsmenge
- V: Menge von Zustandsvariablen
- A: Menge von Events
- C: Menge der Uhren
- $E \subseteq P \times \mathcal{L}(V \cup V' \cup A \cup C) \times \mathcal{P}(C) \times P$: Transitionsrelation
- $s : P \rightarrow \mathcal{L}(V)$: Invarianten-Prädikat für Zustände
- $I : P \rightarrow \mathcal{L}(C)$: Uhreninvariante für Zustände
- $Init \subseteq \mathcal{L}(V) \times P$: Initiale Transitionen.

Lizenz

Das Systemspezifizierungswerkzeug Syspect wird unter der GNU General Public License [FSF06b], kurz GPL, veröffentlicht werden. Hierfür waren folgende Fakten entscheidend:

- Die PEA Tools von Jochen Hoenicke und die Community Z Tools stehen unter der GPL. Wird eine unter der GPL stehende Bibliothek verlinkt, so muss das verlinkende Programm auch unter der GPL stehen. Bei der Benutzung der oben genannten Bibliotheken muss Syspect also unter der GPL stehen.
- Syspect ist ein Eclipse Plug-In und nutzt des Weiteren das Eclipse Plug-in GEF [EFI06c]. Diese und Eclipse selbst stehen unter der Eclipse Public License, welche eine freie Lizenzwahl des verlinkenden Plug-Ins erlaubt, sofern sich dieses nicht aus Eclipse Public License lizensierten Applikationen ableitet. Syspect selbst besteht nicht aus abgeleiteter Arbeit (im amerikanischen Recht „derivative work“ genannt) und kann somit jede beliebige Lizenz annehmen.

Aufgrund der restriktiven Bedingungen der GPL wird Syspect unter die GPL gestellt.

Richtlinien für die Fehlerbehandlung

Fehlerbehandlung

In diesem Abschnitt wird allgemein beschrieben, wie dem Nutzer mitgeteilt werden kann, dass ein Fehler im Programm aufgetreten ist. Es wird insbesondere nicht darauf eingegangen, wie diese Fehler im Programm entdeckt werden können.
Ist ein kritischer Fehler aufgetreten, so muss dieser dem Nutzer mitgeteilt werden. Dies geschieht auf mindestens zwei verschiedenen Wegen, nämlich per Dialog so wie per Logdatei. Die folgenden Informationen müssen dabei in beiden Meldungen vorhanden sein:

- Eine aussagekräftige Fehlermeldung
- Provider Name
- Plug-in Name (user friendly name)
- Plug-in ID
- Version

Fehlerdialog

Dem Benutzer wird mit Hilfe eines Fehlerdialogs mitgeteilt, dass ein Fehler aufgetreten ist. Dieser muss die oben beschriebenen Informationen enthalten. Zusätzlich ist ein aussagekräftiger Dialogtitel zu wählen. Der Dialog wird mit der `JFace` Klasse `ErrorDialog` erzeugt. Damit ergibt sich dann ein Aussehen wie etwa in Abbildung 3.

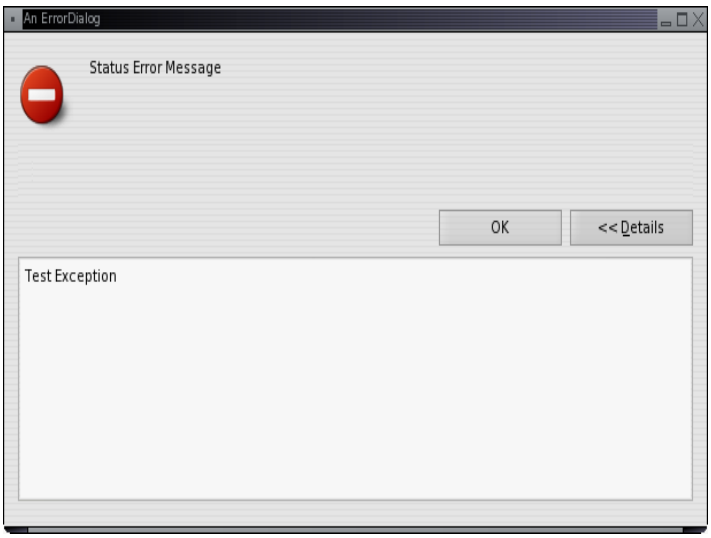


Abbildung 3: Beispiel eines Fehlerdialogs

Logdatei

Weiterhin muss jeder aufgetretene Fehler auch in die Eclipse Logdatei geschrieben werden. Auch hierzu gibt es bereits einfach nutzbare Schnittstellen (via Plugin.getLog().log(...)). Die so geschriebene Nachricht wird von Eclipse in eine im Workspace im Unterverzeichnis `.metadata` liegenden Logdatei geschrieben. Dadurch erhält der Nutzer die Möglichkeit, die Fehlermeldung später wiederzufinden und zur Fehlerbehebung zu nutzen.

Glossar

Assoziation (UML) (Fast) alle Verbindungen bzw. Beziehungen in UML heißen Assoziationen. Die konkreten Instanzen von Assoziationen (also das, was Objekte für Klassen sind) heißen Links.

Aggregation ist eine spezielle Assoziationen, die eine *Teil-vom-Ganzen*-Beziehung darstellt. Die *Teile* des *Ganzen* können auch einzeln bestehen im Vergleich zu einer Komposition.

Capsule ist ein Stereotyp für eine Klasse im CSP-OZ-DC-UML-Profil.

Code Completion Autovervollständigung für Anweisungen/Befehle/Objekte in Code-Editoren.

Community-Z-Tools (CZT) ist eine Ansammlung von Werkzeugen für die formale Spezifikationssprache Z.

Countertrace DC-Countertraces sind spezielle DC-Formeln.

CSP steht für **C**ommunicating **S**equential **P**rocesses und ist eine Spezifikationssprache zur Beschreibung kommunizierender Prozesse, die von Tony Hoare entwickelt wurde [Hoa04].

CSP-OZ-DC Dies ist eine Kombination der drei Spezifikationssprachen CSP [Hoa04], Object-Z [Smi00] und Duration Calculus [ZHR91] zur Spezifikation von Abläufen, Daten und Zeit.

CSP-OZ-DC-Spezifikation Das sind mehrere CSP-OZ-DC-Klassen, die zusammen das Verhalten eines Systems spezifizieren. Der Spezifikations-Begriff wird hier anders verwendet als in der UML.

DC Der **D**uration **C**alculus, entwickelt von Zhou, Hoare und Ravn, ist eine Intervall Temporal Logik für Realzeitsysteme.

Eclipse ist eine Open-Source-Community, deren Projekte darauf ausgerichtet sind, eine erweiterbare Entwicklungs-Plattform und Anwendungssysteme zur Software-Entwicklung bereitzustellen [EFI05b].

EMF Das **E**clipse **M**odeling **F**ramwork ist ein Open Source Java-Framwork zur automatisierten Erzeugung von Quelltext anhand von strukturierten Modellen.

Entwurfsmuster (engl. design pattern) beschreibt eine bewährte Schablone für ein Entwurfsproblem. Es stellt damit eine wiederverwendbare Vorlage, unabhängig von einer Programmiersprache, zur Problemlösung dar.

Extension Point ist eine von einem Plug-in angegebene Erweiterungsmöglichkeit.

Factory ist eine Klasse, die nach bestimmten Kriterien Objekte anderer Klassen erzeugt.

Feature steht allgemein in der Software-Entwicklung für eine besondere Eigenschaft/Fähigkeit eines Produktes. Im Eclipse-Umfeld gibt es jedoch eine weitere Bedeutung: Ein Feature ist hier eine Möglichkeit verschiedene Plug-ins und ihre Fähigkeiten zu gruppieren. Es sind XML-Dateien, die keinen Code enthalten, sondern nur die Informationen über die von einem Produkt benötigten Plug-ins und wie erforderliche Updates möglich sind.

GEF Das **G**raphical **E**diting **F**ramework ist ein Plug-in, das ein Framework für die Erstellung von grafischen Editoren liefert.

Generalisierung ist in der UML eine gerichtete Beziehung zwischen einem *generelleren* und einem *spezielleren* Classifier.

GUI **G**raphical **U**ser **I**nterface (Grafische Benutzeroberfläche). Das ist eine grafische Schnittstelle, um Interaktion mit dem Benutzer zu ermöglichen.

JVM steht für **J**ava **V**irtual **M**achine. Sie ist der Teil der Java-Laufzeitumgebung (JRE), der für die Ausführung des Java-Bytecodes verantwortlich ist.

LaTeX Lamports TeX, entwickelt von Leslie Lamport (1984), ist ein Softwarepaket, dass die Benutzung des Textsatzprogramms TeXvereinfacht.

Kommentar ist eine Anmerkung, die ein Modellierer in einem UML-basierten Modell einfügen kann. Ein Kommentar kann mit anderen UML-Elementen verbunden sein.

Komposition ist eine spezielle Assoziationen, die eine *Teil-vom-Ganzen*-Beziehung darstellt. Die *Teile* des *Ganzen* können nicht einzeln bestehen im Vergleich zu einer Aggregation.

Konsistenz bedeutet Zusammenhalt und meint hier das schlüssige Zusammenpassen von View und Model ohne das Widersprüche auftreten.

Model-Checker Ein Model-Checker ist eine Software zur automatischen Verifikation von Systembeschreibungen (Modellen) gegen eine Spezifikation.

MOF **M**eta-**O**bject **F**acility ist die Metadaten-Architektur der OMG.

Object-Z Dies ist eine objektbasierte Spezifikationssprache [Smi00] zur Beschreibung von Daten, Zuständen und deren Transformationen.

OMG **O**bject **M**anagement **G**roup, ein Konsortium für die Entwicklung von Standards für die herstellerunabhängige systemübergreifende objektorientierte Programmierung. Mitglieder sind unter anderen IBM, Apple und Sun. Die bekanntesten Entwicklungen sind CORBA und UML.

PEA **P**hasen **E**vent **A**utomat. Ein spezielles Timed Automata Modell, das als operationelle Semantik für CSP-OZ-DC definiert wurde [Hoe06].

PEA-Tool Tool von Jochen Hönicke und Roland Meyer zur Darstellung von Phasen Event Automaten als Java-Repräsentationen.

Plug-in ist ein Zusatz-Modul, das über vordefinierte Schnittstellen an ein Software-Produkt angeschlossen werden kann und dessen Funktionalität erweitert.

RCP Die Eclipse **R**ich **C**lient **P**latform ist ein Sammlung von Plug-ins zur Entwicklung einer Rich Client Anwendung.

Region ist eine UML-Region in Zustandsautomaten.

Stereotype ist in der UML 2.0 eine Metaklasse, die verwendet wird, um die Semantik von UML-Basiselementen (wie z.B. Klassen) genauer zu spezifizieren.

Swing API zum Programmieren von grafischen Benutzeroberflächen. Swing wurde von Sun Microsystems für die Programmiersprache Java entwickelt.

SWT Das **S**tandard **W**idget **T**oolkit ist eine Bibliothek für die Erstellung grafischer Oberflächen mit Java.

Syntax-Highlighting Darstellung von Wörtern und Zeichenkombinationen abhängig von der Bedeutung in unterschiedlichen Farben, Schriftarten und -stilen.

Tagged Values sind Standard-Metaattribute von UML-Stereotypen.

Testformeln sind eine Teilklasse aller DC-Formeln [Hoe06]. Eine Testformel ist die Negationen einer Formel, die darauf getestet werden soll, ob ein gegebener PEA sie erfüllt.

TEX Textsatzsystem mit integrierter Makrosprache.

UML **U**nified **M**odeling **L**anguage, eine standardisierte Beschreibungssprache, um Strukturen und Abläufe in objektorientierten Softwaresystemen darzustellen.

UML-Diagramm grafische Notationsweise, die eine bestimmte Sicht auf ein mit UML modelliertes System bietet.

UML-Modell Wenn man ein Software-System mit Hilfe von UML modelliert, so ist die Gesamtheit dieser Diagramme das UML-Modell dieses Software-Systems. Im Gegensatz zu einer CSP-OZ-DC-Spezifikation kann ein UML-Modell unvollständig sein bzw. redundante Informationen enthalten, da es sich um eine Sammlung von verschiedenen Sichten handelt. Wenn wir bei *Syspect* eine CSP-OZ-DC-Spezifikation mit Hilfe des UML-Profils mit UML-Diagrammen modellieren, so

wollen wir die Gesamtheit dieser Diagramme, die diese CSP-OZ-DC-Spezifikation abbilden, UML-Modell der CSP-OZ-DC-Spezifikation nennen.

UML-Spezifikation Das ist das Dokument, was die UML (bzw. eine bestimmte Version davon) definiert. Der Spezifikations-Begriff wird hier anders verwendet als bei CSP-OZ-DC.

UML-Profil ist eine konkrete Erweiterung der UML.

Wizard ist eine GUI, die dem Benutzer - meistens in Dialogform - eine besondere Hilfestellung bei der Dateneingabe in Bezug auf ein bestimmtes Ziel gibt.

XMI **X**ML **M**etadata **I**nterchange, Mitglied der XML-Sprachfamilie. Es ist ein Standard der OMG, insbesondere für Datenaustausch zwischen UML-basierten Software-IDEs. Es kann aber alles ausgedrückt werden, was auf MOF basiert.

XML E**x**tensible **M**arkup **L**anguage, bietet eine einfache Syntax, mit der die Struktur von Daten beschrieben werden kann. XML ist damit eine Metasprache mit der andere Sprachen beschrieben werden können.

ZML XML-Format für die Spezifikationssprache Z

Index

LaTeX, 47, 109, 152
LaTeX-Quellcode, 105, 111

Actions, 161
Aggregation, 108
Assoziation, 107, 114, 210

base, 107, 108

capsule, 107, 108, 113
Code Completion, 114
Commands, 149
Community-Z-Tools, 43, 148, 151
conjugated, 107, 108
Countertraces, 114
CSP, 57, 69, 199
CSP-OZ-DC, 69, 74, 77, 105, 109, 111, 119, 151
CSP-OZ-DC-LaTeX, 106
CSP-OZ-DC-XML, 106
CSP-Prozess, 58, 105

data, 107
Dateiformat, 165
dc-counterexample, 113
DeleteActionHook, 162
Drag and Drop, 164
Draw2d, 149
Druckfunktion, 111
Duration Calculus, 69, 71, 73, 151

Eclipse, 7, 17
Editparts, 149
Editpolicies, 149
EMF, 209
Endzustand, 109, 176
Export, 148
Exportfunktionen, 109
Extension Point, 152

Factory, 149, 210
Formeleditor, 107, 114

GEF, 25, 148, 149
Generalisierung, 107
GMF, 148, 151

init, 113
Initital-Zustand, 109
Instanz einer Klasse, 108
invariant, 113

Klasse, 107, 113
Klassendiagramm, 50, 52, 78, 105, 107, 109, 111, 112
Kommentar, 109
Komponentendiagramm, 176, 196
Komposition, 107
Konsistenz, 107

Metadaten, 112
Model-Checker, 105, 109, 111
Model-Checking, 74
Model-Ebene, 168, 174
MOF, 49, 209
MVC, 157

Object-Z, 43, 57, 69, 73, 151
Observer, 149

PEA-Tool, 148, 151
PEA-XML, 106
Performance, 118
Phasen-Event-Automaten, 72, 105, 109, 111, 151, 219
Plug-in, 17, 148, 152, 153
Port, 108, 117
Preference, 162
protocol, 107, 113

Rich Client Platform, 148, 152, 159

Schablone, 113
Startzustand, 176
Statemachine, 79
Stereotyp, 107, 113, 210
Strukturdiagramm, 50, 79, 105, 108, 111, 116
Superzustand, 109
SWT, 149

Tabbed Property View, 190
Tags, 107, 113
Testformeln, 114
Transition, 109, 151, 210
Typprüfung, 114

UML, 49, 54
UML-Modell, 105, 111
UML-Profil, 77, 105, 111
UML2, 210
Unicode-Darstellung, 114

View-Ebene, 169, 174
Viewable-Ebene, 169, 174

XMI, 54, 109, 210
XML, 54, 109, 119, 151, 209
XML-Darstellung, 105, 111

ZML, 44, 151, 152
Zustand, 109, 175
Zustandsautomat, 175
Zustandsdiagramm, 51, 52, 105, 107–109, 111, 115, 175

Abbildungsverzeichnis

1.1 Project Wizard 9
1.2 Java-Perspektive 10
1.3 Debug-Perspektive 11

2.1 Struktur von Eclipse [Sha04] 18
2.2 Screenshots des Update-Managers 19
2.3 extension-point und extension 21
2.4 Plugin.xml und Implementierung 21
2.5 Eine Manifest-Datei eines Plug-ins und dessen Java-Klasse 22
2.6 Schnittstellen der Resourcen [Sha04] 22
2.7 Architektur einer Ansicht [Sha04] 24
2.8 Features [Sha04] 25
2.9 Beispiel für einen grafischen Editor mit GEF 25
2.10 GEF-Architektur 26

3.1 Das Model-View-Controller Konzept 28
3.2 Die Implementierung des MVC-Modells in FUJABA [Tic05] 29
3.3 Das FUJABA zugrunde liegende Meta-Modell [Tic05] 30
3.4 Die FUJABA4Eclipse Struktur [Mey05a] 31

4.1 Benutzeroberfläche ArgoUML 35

6.1 Ein Kompositionsstrukturdiagramm am Beispiel einer Wetterstation . . 52
6.2 Ein Klassendiagramm am Beispiel einer Türsteuerung 52
6.3 Protokollzustandsdiagramm am Beispiel einer Türsteuerung 53
6.4 Verhaltenszustandsdiagramm am Beispiel einer Ampelsteuerung 53

7.1 Beispiel für ein reaktives System 58
7.2 Der CSP-Prozess *STOP* 59
7.3 Der CSP-Prozess *SKIP* 59
7.4 Der CSP-Prozess $knopf1 \rightarrow fahreZu1 \rightarrow STOP$ 59
7.5 CSP-Prozess mit $\Box$ 60
7.6 Der CSP-Prozess *FS* 60
7.7 Der CSP-Prozess P_4 61
7.8 Der CSP-Prozess P_5 61
7.9 CSP-Prozess in Trace-Semantik 62
7.10 CSP-Prozess in SOS 63

7.11 Z-Schema Fahrstuhl 64
7.12 Dekoriertes Z-Schema Fahrstuhl 64
7.13 Z-Schema der Operation *fahreZu* 65
7.14 Z-Schema der Operation *fahreZu* mit Δ-Operator 65
7.15 Z-Schema der Operation *fahreZu* 65
7.16 CSP-OZ-Schema 66
7.17 CSP-OZ-Schema eines Fahrstuhls 66

8.1 Erfüllende Interpretation für $\lceil P \rceil ; \lceil Q \rceil$ 72
8.2 CSP-OZ-DC Spezifikation eines Zählers 74
8.3 Automat zum CSP-Teil 75
8.4 Automat zum Object-Z-Teil 75
8.5 Automaten zum DC-Teil 76

9.1 Klassendiagramm der *Bank* inklusive der benötigten Protokolle (aus [MORW05]) 80
9.2 Statemachine der *Bank* (aus [MORW05]) 81
9.3 Strukturdiagramm der Bankverwaltung (aus [MORW05]) 82

10.1 Klassenstruktur unter JUnit 85
10.2 Klassenstruktur unter JUnit 88
10.3 Generierung von JUnit-Testklassen aus JML-Dateien 91

12.1 Schematische Übersicht der Entwicklungsumgebung 106
12.2 Schematische Übersicht über die Exportfunktionen 110

13.1 Elemente von Zustandsdiagrammen 115
13.2 Diagrammelemente eines Strukturdiagramms 117
13.3 Aufbau einer CSP-OZ-DC-Klasse 119
13.4 XML-Beispiel einer CSP-OZ-DC-Klasse 120
13.5 Aktuelles XML-Format für Phaseneventautomaten 123
13.6 Allgemeine Anwendungsfälle 124
13.7 Anwendungsfälle in der Klassendiagrammansicht 129
13.8 Anwendungsfälle in der Strukturdiagrammansicht 133
13.9 Anwendungsfälle in der Zustandsdiagrammansicht 138

14.1 Klassendiagramm für die Bank nach dem alten/neuen UML-Profil . . . 144
14.2 Klassendiagramm für die ATM nach dem alten/neuen UML-Profil . . . 144
14.3 Strukturdiagramm/Komponentendiagramm des Systems nach dem alten/neuen UML-Profil 145

15.1 Abhängigkeiten des Syspect Werkzeuges 148
15.2 Wesentliche Abläufe des GEF-Frameworks 150
15.3 Abhängigkeiten von Library-Plug-ins 153
15.4 Extension Points in Syspect 154
15.5 Extension Points des Plug-ins `org.eclipse.ui` 155
15.6 Von Syspect Core erweiterte Extension Points 159
15.7 Bereitstellen von `Actions` über Extension Point 161

15.8 *IActionDelegates*-Struktur 162
15.9 Die Verberbungshierarchie von `AbstractSyspectWizard` 166
15.10Eltern-Kind-Beziehungen im *Syspect* Model (in Pfeilrichtung gelesen ist die Semantik „ist Kind von“) 169
15.11Ein in einer „ConstrainedNode“-Klasse aufgetretener Event wird an die Modelebene weitergeleitet (vertikale Richtung) und in der Parent/Child-Hierarchie zur Wurzel propagiert (horizontale Richtung). 170
15.12Superinterfaces 171
15.13Projektmanager und Projekte 172
15.14Klassen und Interfaces 173
15.15Z-Elemente 174
15.16Verbindungen 175
15.17Zustandsautomaten 176
15.18Komponentendiagramme 177
15.19Abstrakte Oberstruktur der *Viewable*-Ebene 177
15.20Kontext: Diagramme und Diagramminhalt 178
15.21Diagramme und Verbindungen auf *Viewable*-Ebene 178
15.22Die Vererbungshierarchie von `ViewableNode` 179
15.23Die *View*-Ebene 179
15.24Vererbungshierarchie bei den CreateCommands 181
15.25Bereitstellung von `ClassTitle` und `Separator` 184
15.26Die Erstellung eines Provided Interfaces bestehend aus einer Figure und einer Verbindungslinie 185
15.27Der Zeichenbereich einer Komponente rot hinterlegt. 186
15.28Der AssemblyConnector macht keinen Unterschied zwischen Provided oder Required Interfaces 187
15.29*Tabbed Property View* 191
15.30*Property Tabs* Extension Point 192
15.31*Property Sections* Extension Point 193
15.32Section-Klassen 194
15.33Widget Factory 194
15.34CSP-OZ-DC-Spezifikation 195

17.1 EMF vereint Java, UML und XML 209
17.2 Assembly 213
3 Beispiel eines Fehlerdialogs 221

Literaturverzeichnis

[Apa06] Apache Software Foundation. The Apache ANT Project (homepage). http://ant.apache.org/, January 01, 2006.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CSFP05] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion. http://svnbook.red-bean.com/en/1.1/svnbook.html, October 2005. For Subversion 1.1 (book compiled from Revision 1337).

[CZT05] Community z tools. `http://czt.sourceforge.net/`, November 02, 2005.

[Doh03] Thomas Dohmke. Subversion - ein Versionskontrollsystem. http://thomas.dohmke.de/de/artikel/subversion, June 2003.

[EFIa] Eclipse Foundation Inc. Eclipse platform plug-in developer guide - message bundles. `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/misc/message_bundles.html`.

[EFIb] Eclipse Foundation Inc. Eclipse Project. `http://www.eclipse.org/eclipse`.

[EFIc] Eclipse Foundation Inc. Eclipse Technology Project. `http://www.eclipse.org/technology`.

[EFId] Eclipse Foundation Inc. Eclipse Tools Project. `http://www.eclipse.org/tools`.

[EFI05a] Eclipse Foundation Inc. Eclipse wiki. `http://eclipsewiki.editme.com`, November 05, 2005.

[EFI05b] Eclipse Foundation Inc. Homepage der Eclipse-Community, November 05, 2005. `http://www.eclipse.org`.

[EFI06a] Eclipse Foundation Inc. Eclipse plugin resource center and marketplace, February 23, 2006. `http://www.eclipseplugincentral.com`.

[EFI06b] Eclipse Foundation Inc. Eclipse plugins, February 23, 2006. `http://www.eclipse-plugins.info`.

[EFI06c] Eclipse Foundation Inc. GEF Eclipse Plug-in, January 31, 2006. http://www.eclipse.org/gef.

[EFI06d] Eclipse Foundation Inc. A shape diagram editor, September 17, 2006. http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html.

[eNo06] eNode. Model-view-controller pattern. http://www.enode.com/x/markup/tutorial/mvc.html/, March 27, 2006.

[Fab05] J. Faber. ETCS Case Study EM: Treatment of Emergency Messages. http://csd.informatik.uni-oldenburg.de/~jfaber/dl/EM_description.pdf, October 27, 2005.

[FB05] Karl Fogel and Moshe Bar. Open Source Development with CVS. http://cvsbook.red-bean.com/cvsbook.html, October 2005. 3rd Edition.

[FDG05] Fujaba Developer Group. The Fujaba API, http://wwwcs.uni-paderborn.de/cs/fujaba/documents/developer/Fujaba/index.html. Website, fujaba.de, October 21, 2005.

[FDR] Failures divergence refinement model checker. http://www.fsel.com.

[FH06] Dr. Bernhard Rumpe Florian Hözl, Markus Pister. Merkblatt JML. http://www4.in.tum.de/lehre/vorlesungen/sw_engineering/WS0203/uebung/merkblattJML.pdf, January 01, 2006.

[Fis00] Clemens Fischer. Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis, Bericht Nr. 2/2000, 2000.

[FOW01] C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP View on UML-RT structure diagrams. Technical report, Department für Informatik, Universität Oldenburg, 2001.

[FSF06a] Free Software Foundation. GNU make. http://www.gnu.org/software/make, April 02, 2006.

[FSF06b] Free Software Foundation. GNU General Public License, January 31, 2006. http://www.fsf.org/licensing/licenses/gpl.html.

[GDB02] Timothy J. Grose, Gary C. Doney, and Stephen A. Brodsky. *Mastering XMI*. Wiley, 2002.

[GHJ95] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[HM05] J. Hoenicke and P. Maier. Model-Checking of Specifications Integrating Processes, Data and Time. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 465–480. Springer, 2005.

[Hoa04] C. A. R. Hoare. Communicating Sequential Processes. http://www.usingcsp.com/cspbooks, 2004.

[Hoe05] J. Hoenicke. private communications, November 2005.

[Hoe06] J. Hoenicke. Combination of processes, data, and time. Dissertation, Department für Informatik, Abt. Entwicklung korrekter Systeme, Universität Oldenburg, July 12, 2006.

[JML06] Java modelling language hompage. `http://www.cs.iastate.edu/~leavens/JML/index.shtml`, January 01, 2006.

[JNT06] The apache ant project (homepage), optionale tasks unter ant - junit. http://ant.apache.org/manual/OptionalTasks/junit.html, January 01, 2006.

[JW96] J. Davies J. Woodcook. Using Z. `http://www.usingz.com`, 1996.

[Kli05] M. Klink. ArgoUML Cookbook. `http://argouml.tigris.org/documentation/defaulthtml/cookbook/`, March 10, 2005.

[Lem06] Simon Lembke. Ant - Das Apache Build-Werkzeug für Java. http://www.antmake.de/, January 01, 2006.

[Mey05a] Matthias Meyer. HowTo extend Fujaba4Eclipse, `http://www.se.eecs.uni-kassel.de/%7Efujabawiki/images/c/c5/HowToExtendFujaba4Eclipse.pdf`. Pdf document, fujaba.de, October 21, 2005.

[Mey05b] R. Meyer. Model-Checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten. `http://csd.informatik.uni-oldenburg.de/~skript/pub/diplom/meyer05.ps.gz`, August 9, 2005.

[Mic97] Sun Microsystems. Java code conventions, 1997. `http://java.sun.com/docs/codeconv/CodeConventions.pdf`.

[MOF06] OMGs MetaObject Facility. `http://www.omg.org/mof/`, March 27, 2006.

[MORW04] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 267–286. springer, March 2004.

[MORW05] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a Formal Method into a Software Engineering Process with UML and Java. unpublished, 2005.

[Old05] E.-R. Olderog. Kombination von Spezifikationstechniken. Vorlesungsskript Sommersemester 2005, 2005.

[PBG04] Torsten Posch, Klaus Birken, and Michael Gerdom. *Basiswissen Softwarearchitektur*. Dpunkt, 2004.

[RHQ+05] Chris Rupp, Jürgen Hahn, Stefan Queins, Mario Jeckle, and Barbara Zengler. *UML 2 glasklar*. Carl Hanser, München, 2005.

[Ros97] A.W. Roscoe. The theory and practice of concurrency, 1997.

[RW03] H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams: Classes and State Machines. Technical report, Department für Informatik, Universität Oldenburg, 2003.

[Sch04] Prof. Dr. Stefan Schöf. Skript Software Engineering. Fachhochschule Oldenburg/Ostfriesland/Wilhelmshafen, 2004.

[Sch05] Prof. Dr. W. Schäfer. FUJABA Dokumenatation Guided Tour, `http://www.se.eecs.uni-kassel.de/%7Efujabawiki/images/2/20/FujabaDoc.pdf`. Pdf document, Universität Paderborn, October 21, 2005.

[Sha04] Sherry Shavour. *Eclipse - Anwendungen und Plug-ins mit Java entwickeln*. Addison-Wesley, 2004.

[Smi00] Graeme Smith. The object-z specification language. Kluwer Academic Publishers, 2000.

[SMT06] C. M. Sperberg-McQueen and Henry Thompson. W3c xml schema. `http://www.w3.org/XML/Schema`, March 23, 2006.

[Sta06] Andreea Stamer. Integration von CSP-OZ in die OO-Softwareentwicklung für die automatische Verifikation. Diplomarbeit, Department für Informatik, Abt. Entwicklung korrekter Systeme, Universität Oldenburg, March 20, 2006.

[Tic05] Matthias Tichy. The Fujaba Diagram Visualization Architecture, `http://www.se.eecs.uni-kassel.de/%7Efujabawiki/images/0/0a/FujabaDiagramVisualization.pdf`. Pdf document, fujaba.de, October 21, 2005.

[tig] tigris.org. Subclipse: Das Eclipse Plug-in für Subversion. `http://subclipse.tigris.org`.

[tig05] tigris.org. ArgoUML Homepage. `http://argouml.tigris.org`, March 10, 2005.

[UML05] Object Managment Group. *Unified Modelling Language*, October 04, 2005. `http://www.uml.org/`.

[Wen05] Lothar Wendehals. 10 Steps To Build a Fujaba Plug-in, `http://www.se.eecs.uni-kassel.de/%7Efujabawiki/images/2/2c/10_Steps_To_Build_a_Fujaba_Plugin.pdf`. Pdf document, fujaba.de, October 21, 2005.

[Wes05] Frank Westphal. Testgetriebene Entwicklung mit JUnit & FIT: Wie Software änderbar bleibt. dpunkt.verlag, 2005.

[Wik05a] Wikipedia. `http://de.wikipedia.org/wiki/Versionsverwaltung`, October 2005.

[Wik05b] WikiPedia. Unified modelling language. `http://de.wikipedia.org/wiki/Unified_Modeling_Language`, November 01, 2005.

[Wik05c] WikiPedia. Bsd lizenz. `http://de.wikipedia.org/wiki/BSD-Lizenz`, November 04, 2005.

[Wik05d] WikiPedia. Kognition. `http://de.wikipedia.org/wiki/Kognition`, October 29, 2005.

[YC06] Gary T. Leavens Yoonsik Cheon. The jml and junit way of unit testing and its implementation. `http://smv.unige.ch/tiki-download_file.php?fileId=533`, January 01, 2006.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *IPL*, 40(5):269–276, 1991.